**Основные элементы интерфейса среды MS SQL Server. Особенности работы в среде MS SQL Server**

**SQL Server Management Studio (SSMS)** – это бесплатная графическая среда, включающая набор инструментов для разработки сценариев на T-SQL и управления инфраструктурой Microsoft SQL Server.

Среда SQL Server Management Studio – это основной, стандартный и полнофункциональный инструмент для работы с Microsoft SQL Server, разработанный компанией Microsoft, который предназначен как для разработчиков, так и для администраторов SQL Server.

С помощью SSMS Вы можете разрабатывать базы данных, выполнять инструкции T-SQL, а также администрировать Microsoft SQL Server.

Основной функционал SSMS:

1. Подключение к службам SQL Server
2. Обозреватель объектов
3. Обозреватель шаблонов
4. Редактор SQL кода
5. Просмотр плана выполнения запроса
6. Обозреватель решений
7. Конструктор таблиц
8. Конструктор баз данных (Диаграммы баз данных)
9. Конструктор запросов и представлений

С помощью SQL Server Management Studio мы можем подключаться не только к ядру СУБД, но и к другим компонентам SQL Server, например, к службам Analysis Services (SSAS), Integration Services (SSIS) и Reporting Services (SSRS).

Таким образом, мы можем управлять объектами служб Analysis Services, например, выполнять их резервное копирование и обработку, создавать скрипты для служб Analysis Services и выполнять другие операции.

Также используя Management Studio мы можем управлять пакетами SSIS, например, выполнять импорт и экспорт пакетов.

Кроме этого SSMS позволяет администрировать службы Reporting Services, т.е. управлять ролями, заданиями и так далее.

**Обозреватель объектов среды SQL Server Management Studio** – это графический пользовательский интерфейс для просмотра и управления объектами в каждом экземпляре SQL Server.

Обозреватель объектов Management Studio предоставляет интерфейс, в котором структура объектов сервера представлена в очень удобном иерархическом виде, напоминает что-то вроде файловой системы Windows, т.е. папки, подпапки, файлы и т.д.

**Шаблоны в SQL Server Management Studio** – это файлы, содержащие стандартный SQL код, который предназначен для создания объектов на SQL Server.

Иными словами, это готовые SQL скрипты, содержащие заголовки SQL инструкций, в которые всего лишь необходимо подставить свои данные, например, имена объектов, чтобы создать тот или иной объект на SQL Server.

Обозреватель шаблонов Management Studio позволяет просматривать и использовать доступные шаблоны.

Открыть «Обозреватель шаблонов» можно из меню «Вид -> Браузер шаблонов».

Таким образом, мы можем использовать шаблоны для создания или изменения таких объектов как:

Базы данных

Таблицы

Представления

Индексы

Функции

Хранимые процедуры

Триггеры

И другие объекты SQL Server

*Редактор SQL кода*

Основное назначение среды SQL Server Management Studio – это, конечно же, разработка, выполнение и отладка кода на T-SQL, иными словами, написание и выполнение SQL запросов и инструкций. Поэтому SSMS обладает современным и продвинутым редактором SQL кода, который поддерживает технологию IntelliSense, т.е. автодополнение, например, Вы начинаете писать первые буквы объекта, а редактор сам дописывает его, точнее, показывает возможные варианты окончания.

Кроме этого у редактора есть подсветка синтаксиса и другие полезные возможности.

Чтобы открыть редактор SQL кода в среде Management Studio, необходимо на панели инструментов нажать на кнопку «Создать запрос». Именно здесь пишутся и выполняются все SQL запросы и инструкции к базам данных.

Конструктор таблиц

Конструктор таблиц – это визуальный инструмент для разработки таблиц в базах данных.

С помощью конструктора таблиц SQL Server Management Studio можно создавать, редактировать и удалять:

Таблицы

Столбцы

Ключи

Индексы

Связи

Ограничения

Чтобы запустить *«Конструктор таблиц»,* необходимо щелкнуть правой кнопкой мыши по контейнеру *«Таблицы»* в обозревателе объектов, и выбрать пункт *«Создать -> Таблица»*.

Таким образом, благодаря функционалу конструктора таблиц мы может создавать и редактировать таблицы базы данных, при этом даже не используя язык SQL.

Конструктор баз данных (Диаграммы баз данных)

**Конструктор баз данных** — это визуальный инструмент для разработки баз данных, который позволяет конструировать и визуализировать базу данных, с которой установлено соединение.

Конструктор баз данных можно использовать для создания, редактирования и удаления таблиц, столбцов, ключей, индексов, связей и ограничений.

Кроме этого, конструктор позволяет визуализировать базу данных, т.е. создать ER-диаграмму базы данных. Можно создать как одну, так и несколько диаграмм, иллюстрирующих некоторые или все имеющиеся в ней таблицы, столбцы, ключи и связи.

Чтобы запустить конструктор баз данных и создать диаграмму базы данных, необходимо в обозревателе объектов щелкнуть правой кнопкой мыши по контейнеру «Диаграммы баз данных» и выбрать пункт «Создать диаграмму базы данных».

Конструктор запросов и представлений

Конструктор запросов и представлений – это визуальный инструмент для разработки запросов и представлений.

В данном конструкторе доступно 4 панели:

Панель диаграмм – на ней в графическом виде представлен запрос, т.е. какие таблицы и столбцы задействованы в запросе;

Панель критериев – на ней Вы задаете параметры запроса, например, указываете, какие столбцы будут задействованы в запросе, как отфильтровать данные, как упорядочивать результаты и т.д;

Панель SQL – здесь запрос отображается в виде SQL инструкций, где Вы можете в случае необходимости внести изменения на языке SQL;

Панель результатов – она показывает результаты выполнения запроса.

Таким образом, с помощью данного конструктора можно конструировать запросы к базе данных, при этом используя только мышку.

Просмотр свойств объектов

Среда SQL Server Management Studio позволяет просматривать и изменять свойства объектов SQL Server в обозревателе объектов.

Практически у каждого объекта на SQL Server есть свойства, которые как раз и можно изменить с помощью графических инструментов среды SQL Server Management Studio или просто посмотреть.

Например, для редактирования свойств базы данных необходимо в обозревателе объектов выбрать базу, щелкнуть по ней правой кнопкой мыши и выбрать «Свойства».

**Назначение языка Transact-SQL. Основные элементы Transact-SQL. Типы данных MS SQL Server. Выражения Transact-SQL**

SQL (Structured Query Language) - это универсальный компьютерный язык, применяемый для создания, модификации и управления данными в реляционных базах данных (язык структурированных запросов).

SQL в его исходном виде является информационно-логическим языком, а не языком программирования, но вместе SQL предусматривает возможность его процедурных расширений, с учётом которых язык уже вполне может рассматриваться в качестве языка программирования.

В настоящее время широко распространены следующие спецификации SQL:

|  |  |
| --- | --- |
| Базы данных и спецификации SQL | |
| Тип базы данных | Спецификация SQL |
| Microsoft SQL | Transact-SQL |
| Microsoft Jet/Access | Jet SQL |
| MySQL | SQL/PSM (SQL/Persistent Stored Module) |
| Oracle | PL/SQL (Procedural Language/SQL) |
| IBM DB2 | SQL PL (SQL Procedural Language) |
| InterBase/Firebird | PSQL (Procedural SQL) |

Transact-SQL - это процедурное расширение языка SQL компаний Microsoft. SQL был расширен такими дополнительными возможностями как:

управляющие операторы,

локальные и глобальные переменные,

различные дополнительные функции для обработки строк, дат, математики и т.п.,

поддержка аутентификации Microsoft Windows

Язык Transact-SQL является ключом к использованию SQL Server. Все приложения, взаимодействующие с экземпляром SQL Server, независимо от их реализации и пользовательского интерфейса, отправляют серверу инструкции Transact-SQL.

Элементы синтаксиса

Директивы сценария

Директивы сценария - это специфические команды, которые используются только в MS SQL. Эти команды помогают серверу определять правила работы со скриптом и транзакциями. Типичные представители: GO - сигнализирует SQL-серверу об окончании сценария, EXEC (или EXECUTE) - выполняет процедуру или скалярную функцию.

Комментарии

Комментарии используются для создания пояснений для блоков сценариев, а также для временного отключения команд при отладке скрипта. Комментарии бывают как строковыми, так и блоковыми:

-- - строковый комментарий исключает из выполнения только одну строку, перед которой стоят два минуса.

/\* \*/ - блоковый комментарий исключает из выполнения целый блок команд, заключенный в указанную конструкцию.

Типы данных

Как и в языках программирования, в SQL существуют различные типы данных для хранения переменных:

Числа - для хранения числовых переменных (int, tinyint, smallint, bigint, numeric, decimal, money, smallmoney, float, real).

Даты - для хранения даты и времени (datetime, smalldatetime).

Символы - для хранения символьных данных (char, nchar, varchar, nvarchar).

Двоичные - для хранения бинарных данных (binary, varbinary, bit).

Большеобъемные - типы данных для хранения больших бинарных данных (text, ntext, image).

Специальные - указатели (cursor), 16-байтовое шестнадцатиричное число, которое используется для GUID (uniqueidentifier), штамп изменения строки (timestamp), версия строки (rowversion), таблицы (table).

Примечание. Для использования русских символов (не ASCII кодировки) испольюзуются типы данных с приставкой "n" (nchar, nvarchar, ntext), которые кодируют символы двумя байтами. Иначе говоря, для работы с Unicode используются типы данных с "n".

Примечание. Для данных переменной длины используются типы данных с приставкой "var". Типы данных без приставки "var" имеют фиксированную длину области памяти, неиспользованная часть которой заполняется пробелами или нулями.

Идентификаторы

Идентификаторы - это специальные символы, которые используются с переменными для идентифицирования их типа или для группировки слов в переменную. Типы идентификаторов:

@ - идентификатор локальной переменной (пользовательской).

@@ - идентификатор глобальной переменной (встроенной).

# - идентификатор локальной таблицы или процедуры.

## - идентификатор глобальной таблицы или процедуры.

[ ] - идентификатор группировки слов в переменную.

Переменные

Переменные используются в сценариях и для хранения временных данных. Чтобы работать с переменной, ее нужно объявить, притом объявление должно быть осуществлено в той транзакции, в которой выполняется команда, использующая эту переменную. Иначе говоря, после завершения транзакции, то есть после команды GO, переменная уничтожается.

Объявление переменной выполняется командой DECLARE, задание значения переменной осуществляется либо командой SET, либо SELECT:

Операторы

Операторы - это специальные команды, предназначенные для выполнения простых операций над переменными:

Арифметические операторы: "\*" - умножить, "/" - делить, "%" - модуль от деления, "+" - сложить , "-" - вычесть, "()" - скобки.

Операторы сравнения: "=" - равно, ">" - больше, "<" - меньше, ">=" - больше или равно, "<=" меньше или равно, "<>" - не равно.

Операторы соединения: "+" - соединение строк.

Логические операторы: "AND" - и, "OR" - или , "NOT" - не.

Cистемные функции

Спецификация Transact-SQl значительно расширяет стандартные возможности SQL благодаря встроенным функциям:

Агрегативные функции- функции, которые работают с коллекциями значений и выдают одно значение. Типичные представители: AVG - среднее значение колонки, SUM - сумма колонки, MAX - максимальное значение колонки, COUNT - количество элементов колонки.

Скалярные функции- это функции, которые возвращают одно значение, работая со скалярными данными или вообще без входных данных. Типичные представители: DATEDIFF - разница между датами, ABS - модуль числа, DB\_NAME - имя базы данных, USER\_NAME - имя текущего пользователя, LEFT - часть строки слева.

Функции-указатели- функции, которые используются как ссылки на другие данные. Типичные представители: OPENXML - указатель на источник данных в виде XML-структуры, OPENQUERY - указатель на источник данных в виде другого запроса.

Выражения

Выражение - это комбинация символов и операторов, которая получает на вход скалярную величину, а на выходе дает другую величину или исполняет какое-то действие. В Transact-SQL выражения делятся на 3 типа: DDL, DCL и DML.

DDL (Data Definition Language)- используются для создания объектов в базе данных. Основные представители данного класса: CREATE - создание объектов, ALTER - изменение объектов, DROP - удаление объектов.

DCL (Data Control Language)- предназначены для назначения прав на объекты базы данных. Основные представители данного класса: GRANT - разрешение на объект, DENY - запрет на объект, REVOKE - отмена разрешений и запретов на объект.

DML (Data Manipulation Language)- используются для запросов и изменения данных. Основные представители данного класса: SELECT - выборка данных, INSERT - вставка данных, UPDATE - изменение данных, DELETE - удаление данных.

Управление выполнением сценария

В Transact-SQL существуют специальные команды, которые позволяют управлять потоком выполнения сценария, прерывая его или направляя в нужную логику.

Блок группировки - структура, объединяющая список выражений в один логический блок (BEGIN … END).

Блок условия - структура, проверяющая выполнения определенного условия (IF … ELSE).

Блок цикла - структура, организующая повторение выполнения логического блока (WHILE … BREAK … CONTINUE).

Переход - команда, выполняющая переход потока выполнения сценария на указанную метку (GOTO).

Задержка - команда, задерживающая выполнение сценария (WAITFOR)

Вызов ошибки - команда, генерирующая ошибку выполнения сценария (RAISERROR)

Динамическое конструирование выражений

Обычно базы данных создаются и заполняются с помощью сценариев (скриптов) - хотя визуальный редактор прост в обращении, но им никогда быстро и без недочетов не создашь большую базу данных и не заполнишь ее данными. Если вспомнить начало статьи, то опытная база данных как раз создавалась и заполнялась с помощью сценария. Сценарий - это одно или более выражений, объединенных в логический блок, которые автоматизируют работу администратора.

Обычно сценарии пишутся как универсальное средство для выполнения стандартных задач, поэтому в них применяется динамическое конструирование логики - в запросы и команды вставляются переменные, а не конкретные названия объектов, что позволяет быстро изменять параметры скрипта.

Выборка данных

В языках SQL выборка данных из таблиц осуществляется с помощью команды SELECT:

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <название таблицы>

По умолчанию в команде SELECT используется параметр ALL, который можно не указывать. Если в команде указать параметр DISTINCT, то в результат попадут только уникальные (неповторяющиеся) записи из выборки.

Для того, чтобы изменить имена объектов в командах к SQL-серверу, используется команда AS. Использование этой команды помогает сокращать длину строки запроса, а так же получать результат в более удобочитаемом виде.

Пример:

-- Выбрать все записи из таблицы Local Customers

SELECT \* FROM [Local Customers]

|  |  |  |
| --- | --- | --- |
| CustomerID | CustomerName | CustomerAddress |
| 1 | Alex Company | 606443, Russia, Bor, Lenina str., 15') |
| 2 | Potrovka | 115516, Moscow, Promyshlennaya str., 1 |

Фильтрация данных осуществляется с помощью команды WHERE, в которой используются следующие операторы и команды сравнения: =, <, >, <=, >=, <>, LIKE, NOT LIKE, AND, OR, NOT, BETWEEN, NOT BETWEEN, IN, NOT IN, IS NULL, IS NOT NULL. В общем виде команда SELECT с фильтром выглядит так:

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <название таблицы>

 WHERE <условие>

В строке сравнения разрешается использовать подстановочные символы:

% - любое количество символов;

\_ - один символ;

[] - любой символ, указанный в скобках;

[^] - любой символ, не указанный в скобках.

-- Выбрать все записи из таблицы Users, где DepartmentID = 1

SELECT \* FROM Users WHERE DepartmentID = 1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UserID | UserName | UserSurname | DepartmentID | PositionID |
| 1 | Ivan | Petrov | 1 | 1 |
| 2 | Ivan | Sidorov | 1 | 2 |
| 3 | Petr | Ivanov | 1 | 2 |
| 4 | Nikolay | Petrov | 1 | 3 |

Фильтрация позволяет использовать подзапросы, то есть конструировать запрос из нескольких подзапросов:

Для сортировки данных в выборке используется командаORDER BY, но следует учесть, что эта команда не сортирует данные типа text, ntext и image. По умолчанию сортировка производится по возрастанию, поэтому параметр ASC в этом случае можно не указывать:

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <название таблицы>

 WHERE <условие> ORDER BY <названия колонок> [ASC или DESC]

Для того, чтобы ограничить количество строк в результате запроса, используется командаTOP:

SELECT [ALL или DISTINCT] TOP [количество строк] <названия колонок или \*>

 FROM <название таблицы> WHERE <условие>

 ORDER BY <названия колонок> [ASC или DESC]

Внутри запроса можно проводить вычисления над полученными данными. Для этого используюся функции агрегирования:

AVG(колонка) - среднее значение колонки;

COUNT(колонка) - количество не NULL элементов колонки;

COUNT(\*) - количество элементов запроса;

MAX(колонка) - максимальное значение в колонке;

MIN(колонка) - минимальное значение в колонке;

SUM(колонка) - сумма значений в колонке.

Примеры использования команд ORDER, TOP и функций агрегирования:

-- Выбрать 3 первые уникальные записи колонки UserName из таблицы Users,

-- отсортированных по возрастанию UserName

SELECT DISTINCT TOP 3 UserName FROM Users ORDER BY UserName

|  |
| --- |
| UserName |
| Andrey |
| Ivan |
| Nikolay |

-- Выбрать 15 процентов строк из таблицы Users

SELECT TOP 15 PERCENT \* FROM Users

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UserID | UserName | UserSurname | DepartmentID | PositionID |
| 1 | Ivan | Petrov | 1 | 1 |
| 2 | Ivan | Sidorov | 1 | 2 |

Группировка данных

SQL позволяет производить группировку данных по определенным полям таблицы. Чтобы сгруппировать данные по какому-нибудь параметру, в SQL-запросе необходимо написать команду GROUP BY, в которой указать имя колонки, по которой производится группировка. Колонки, упомянутые в команде GROUP BY, должны присутствовать в команде SELECT, а так же команда SELECT должна содержать функцию агрегирования, которая будет применена к сгруппированным данным.

-- Найти количество работников в каждом отделе (сгруппировать работников по

-- идентификатору отделов и сосчитать количество записей в каждой группе)

SELECT DepartmentID, COUNT(UserID) AS 'Number of users'

 FROM Users GROUP BY DepartmentID

|  |  |
| --- | --- |
| DepartmentID | Number of users |
| 1 | 4 |
| 2 | 3 |
| 4 | 1 |

Чтобы отфильтровать строки в запросе с группировкой применяется специальная команда HAVING, в которой указывается условие фильтрации. Колонки, по которым производится фильтрация, должны присутствовать в команде GROUP BY. Команда HAVING может использоваться и без GROUP BY, в этом случае она работает аналогично команде WHERE, но она разрешает применять в условиях фильтрации только функции агрегирования.

-- Найти количество работников в первом отделе (сгруппировать работников по

-- идентификатору отделов, сосчитать количество записей в каждой группе и

-- вывести в результат только отдел с идентификатором равным 1)

SELECT DepartmentID, COUNT(UserID) AS 'Number of users'

 FROM Users GROUP BY DepartmentID HAVING DepartmentID = 1

|  |  |
| --- | --- |
| DepartmentID | Number of users |
| 1 | 4 |

Команда группировки может дополняться оператором WITH ROLLUP, который дополняет результат группировки сводной строкой с суммой значений колонок.

-- Найти количество работников в каждом отделе (сгруппировать работников по

-- идентификатору отделов и сосчитать количество записей в каждой группе),

-- а также сосчитать общее количество работников

SELECT DepartmentID, COUNT(UserID) AS 'Number of users'

 FROM Users GROUP BY DepartmentID WITH ROLLUP

|  |  |
| --- | --- |
| DepartmentID | Number of users |
| 1 | 4 |
| 2 | 3 |
| 4 | 1 |
| NULL | 8 |

Команда группировки также может дополняться оператором WITH CUBE, который дополняет формирует всевозможные комбинации из группируемых колонок: если есть N колонок, то получится 2^N комбинаций.

-- Найти количество работников с определенной должностью в каждом отделе

-- (сгруппировать работников по идентификатору должностей и отделов и

-- сосчитать количество записей в каждой группе), а также сосчитать

-- количество работников по каждой должности, по каждому отделу и

-- общее количество работников

SELECT DepartmentID, PositionID, COUNT(UserID) AS 'Number of users'

 FROM Users GROUP BY DepartmentID, PositionID WITH CUBE

Функция агрегирования GROUPING позволяет определить, была ли запись добавлена командами ROLLUP и CUBE, или это запись получена из источника данных.

-- Найти количество работников в каждом отделе (сгруппировать работников по

-- идентификатору отделов и сосчитать количество записей в каждой группе)

-- а так же пометить дополнительные строки, несуществующие в источнике данных

SELECT DepartmentID, COUNT(UserID) AS 'Number of users',

 GROUPING(DepartmentID) AS 'Added row'

 FROM Users GROUP BY DepartmentID WITH ROLLUP

Еще одна команда группировки COMPUTE позволяет группировать данные и выводить по ним отчет в разные таблицы. То есть команда GROUP BY с операторами ROLLUP и CUBE группирует данные и дописывает в таблицу дополнительны строки с отчетом, а команда COMPUTE группирует данные, разрывая исходную таблицу на несколько подтаблиц, а также формирует подтаблицы с отчетами. Команда COMPUTE может использоваться в двух режимах:

как простая функция агрегирования, выводящая результат в отдельную таблицу;

с параметром BY как команда группировки, разрезающая таблицу на несколько подтаблиц

Команда COMPUTE с параметром BY может использоваться только совместно с командой ORDER BY, причем столбцы сортировки должны совпадать со столбцами группировки.

-- Вывести таблицу пользователей компании, а также посчитать их количество

SELECT \* FROM Users COMPUTE COUNT(UserID)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| UserID | UserName | UserSurname | DepartmentID | PositionID |
| 1 | Ivan | Petrov | 1 | 1 |
| 2 | Ivan | Sidorov | 1 | 2 |
| 3 | Petr | Ivanov | 1 | 2 |
| 4 | Nikolay | Petrov | 1 | 3 |
| 5 | Nikolay | Ivanov | 2 | 1 |
| 6 | Sergey | Sidorov | 2 | 3 |
| 7 | Andrey | Bukin | 2 | 2 |
| 8 | Viktor | Rybakov | 4 | 1 |

-- Найти количество работников в каждом отделе (сгруппировать работников по

-- идентификатору отделов и сосчитать количество записей в каждой группе)

SELECT \* FROM Users ORDER BY DepartmentID COMPUTE COUNT(UserID) BY DepartmentID

Соединение таблиц

Самые важные и нужные запросы в SQL - это с запросы с соединением таблиц, когда выборка осуществляется сразу из нескольких источников. Такие запросы более сложны в написании, но и более удобны в обработке, так как часто выдают в программу уже готовый результат, который остается только вывести на экран.

Соединять таблицы в SQL можно двумя способами: вертикально и горизонтально.

Вертикальное соединение осуществляется командой UNION, которая в конец первой таблицы допишет вторую таблицую. При таком соединении количество колонок соединяемых таблиц должно быть одинаковым, а сами колонки должны иметь одинаковые названия и типы данных. При соединении одинаковые строки, встречающиеся в обоих таблицах, будут удалены, если в команде не указан параметр ALL.

Горизонтальное соединение производится путем сцепки нескольких таблиц по ключевым колонкам. Самое простое горизонтальное соединение выполняется с помощью команды INNER JOIN, которая сцепляет таблицы, выбирая строки по ключевому полю, которое встречается в обоих таблицах.

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <таблица\_1>

 INNER JOIN таблица\_2 ON таблица\_1.ключевое\_поле = таблица\_2.ключевое\_поле

Чтобы выполнить сцепление по всем полям левой таблицы, независимо, есть ли такие записи в правой таблице, необходимо использовать команду LEFT JOIN. Эта команда соединяет таблицы, выбирая все строки из левой таблицы, а отсутствующие данные правой таблицы заполняются значением NULL.

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <таблица\_1>

 LEFT JOIN таблица\_2 ON таблица\_1.ключевое\_поле = таблица\_2.ключевое\_поле

Команда RIGHT JOIN аналогична предыдущей, разница заключается лишь в том, что она соединяет таблицы, выбирая все строки из правой таблицы, а отсутствующие данные левой таблицы заполняются значением NULL.

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <таблица\_1>

 RIGHT JOIN таблица\_2 ON таблица\_1.ключевое\_поле = таблица\_2.ключевое\_поле

Команда FULL JOIN объединяет в себе левое и правое сцепление, то есть она соединяет таблицы, выбирая строки из обоих таблиц, а отсутствующие данные заполняются значением NULL.

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <таблица\_1>

 FULL JOIN таблица\_2 ON таблица\_1.ключевое\_поле = таблица\_2.ключевое\_поле

Последняя и редкоиспользуемая команда соединения таблиц - это CROSS JOIN. Эта команда сцепляет таблицы без использования ключевого поля, а результат - это комбинация из всевозможных строк исходных таблиц.

SELECT [ALL или DISTINCT] <названия колонок или \*> FROM <таблица\_1>

 CROSS JOIN таблица\_2

Сцепление не ограничивается только двумя таблицами, запрос может содержать несколько команда JOIN, что очень удобно при формировании конечных отчетов. Ниже приведены примеры для всех команд соединения таблиц.

Изменение данных

За транзакции в Transact-SQL отвечает структура BEGIN TRANSACTION ... COMMIТ TRANSACTION. Эту структуру использовать необязательно, но тогда все команды сценария являются необратимыми, то есть нельзя сделать "откат" к предыдущему состоянию. Полная структура блока транзакций:

BEGIN TRANSACTION [имя транзакции]

 [операции]

COMMIТ TRANSACTION [имя транзакции] или ROLLBACK TRANSACTION [имя транзакции]

Для вставки данных в таблицы SQL-сервера используется команда INSERT INTO:

INSERT INTO [название таблицы] (колонки) VALUES ([значения колонок])

Команда INSERT INTO также разрешает указывать вносимые данные не по порядку следования колонок, но в этом случае нужно обозначить используемый порядок колонок.

-- В таблицу Users вставить строку с данными UserID = 9, UserName = 'Nikolay',

-- UserSurname = 'Gryzlov', DepartmentID = 4, PositionID = 2.

Для того, чтобы изменить значение ячейки таблицы, используется команда UPDATE:

UPDATE [название таблицы] SET [имя колонки]=[значение колонок] WHERE [условие]

Обновление (изменение) значений в таблице можно производить безусловно, с условием или с выборкой данных из другой таблицы.

-- Установить всем должностям зарплату в 2000 единиц.

UPDATE Positions SET BaseSalary = 2000

Удаление данных производится командой DELETE:

DELETE FROM [название таблицы] WHERE [условие]

Удаление данных обычно производится по какому-то критерию. Так как удаление данных - это достаточно опасная операция, то перед выполнением такой команды лучше всего произвести тестовую выборку командой SELECT, которая выведет в результат те данные, которые будут стерты. Если это то, что требуется, тогда можно смело заменять SELECT на DELETE и выполнять удаление данных.

Более быстрая команда для очистки таблицы - это TRUNCATE TABLE.

TRUNCATE TABLE [название таблицы]

Пример удаления всех данных:

-- Очистить таблицу Users

TRUNCATE TABLE Users

Transact-SQL позволяет использовать временные таблицы, то есть таблицы, которые создаются в памяти сервера на время работы пользователя с базой данных. Временные таблицы могут иметь любое имя, но начинаться обязаны с символа #.

-- Создать временную таблицу #TempTable, в которую скопировать содержание

-- колонки UserName таблицы Users

SELECT UserName INTO #TempTable FROM Users

-- Выбрать все записи временной таблицы #TempTable

SELECT \* FROM #TempTable

Хранимые процедуры и функции

Хранимые процедуры и функции представляют собой набор SQL-операторов, которые можно сохранять на сервере. Если сценарий сохранен на сервере, то клиентам не придется повторно задавать одни и те же отдельные операторы, вместо этого они смогут обращаться к хранимой процедуре. Ситуации, когда хранимые процедуры особенно полезны:

Многочисленные клиентские приложения написаны на разных языках или работают на различных платформах, но должны выполнять одинаковые операции с базами данных.

Безопасность играет первостепенную роль. Хранимые процедуры используются для всех стандартных операций, что обеспечивает совместимость и безопасность среды, а процедуры гарантируют надлежащую регистрацию каждой операции. При таком типе установки приложения и пользователи не получают непосредственный доступ к таблицам базы данных и могут выполнять только конкретные хранимые процедуры.

Необходимо снизить сетевой трафик между клиентом и сервером. Объем пересылаемой информации между сервером и клиентом существенно снижается, но увеличивается нагрузка на систему сервера баз данных, так как в этом случае на стороне сервера выполняется большая часть работы по обработке данных.

Итак, хранимые процедуры и функции дают следующие преимущества:

производительность;

общая логика для всез запросов;

уменьшение трафика;

безопасность - доступ пользователю дается не к таблице, а к процедуре;

**Наблюдение за БД с целью оценки производительности. Механизм переноса БД. Резервное копирование БД. Восстановление БД из резервной копии.**

**Инструменты для диагностики SQL Server**

**T-SQL** — самый мощный, простой и незаменимый инструмент для поиска проблем и анализом производительности SQL Server. Практически все другие инструменты для работы с SQL Server используют T-SQL. Нет ничего такого, чтобы вы не смогли сделать с помощью T-SQL.

**SQL Server Management Studio** — без SSMS практически невозможно работать с SQL Server. С помощью SSMS вы можете посмотреть Activity monitor, проанализировать план запроса, посмотреть параметры сервера или базы данных и многие другие вещи.

**Журналы ошибок SQL Server и Windows** – если что-то идёт не так, журнал ошибок — это первое место, куда смотрит системный администратор. Журнал ошибок SQL Server можно посмотреть через SSMS. Журналы Windows можно посмотреть через оснастку eventvwr.msc.

**Монитор ресурсов Windows** — resmon.exe незаменимый инструмент Windows для быстрой оценки состояния ресурсов сервера. Использование оперативной памяти и процессора можно посмотреть и через Диспетчер задач, но детальное использование сети и жесткого диска можно посмотреть только через resmon и perfmon.

**Системный монитор Windows (Performance Monitor)** — Perfmon.exe это основное средство мониторинга Windows, он содержит в себе разнообразные “счетчики”, как системных метрик, так и приложений, включая SQL Server.

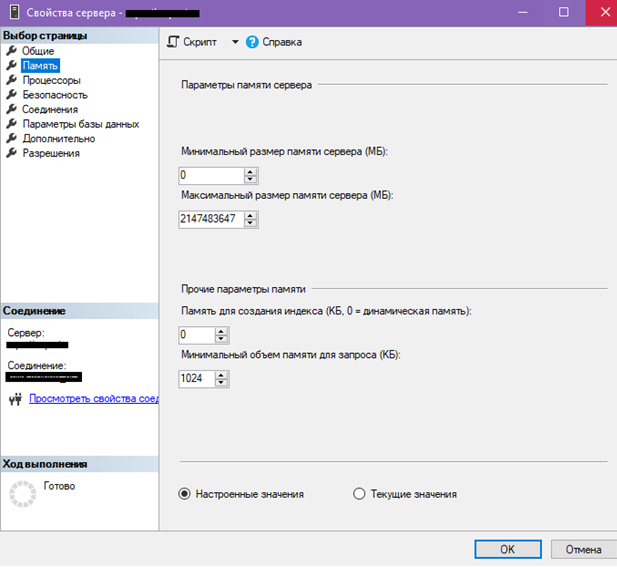
Обнаружение и решение проблем с производительностью SQL Server

Самой распространенной проблемой с которой сталкивается системный администратор, работающий с SQL Server, это жалобы пользователей на производительность запросов и самого сервера: “тормозит”, “долго выполняется запрос“, и так далее.

Прежде всего нужно убедиться, что серверу хватает ресурсов. Рассмотрим, как в SQL Server быстро проанализировать использование памяти, CPU, дисков и наличие блокировок.

Анализ использования оперативной памяти SQL Server

Для начала нужно определить сколько памяти доступно SQL Server. Для этого запустите SSMS (SQL Server Management Studio), зайдите на сервер и зайдите в свойства сервера (ПКМ по названию сервера в Обозревателе объектов).



Сам по себе доступный объём RAM вам ничего не скажет. Нужно сравнить это число с используемой памятью в Диспетчере Задач и самим движком SQL Server с помощью DMV.

В Диспетчере задач, во вкладке Подробности, найдите sqlservr.exe и посмотрите сколько оперативной памяти использует этот процесс.

Если на сервере, например, 128 GB оперативной памяти, а процесс sqlservr.exe использует 60 GB и ограничений по RAM у SQL Server нет, то оперативной памяти вам хватает.

Если SQL Server использует 80-90% RAM от заданной или максимальной, то в таком случае нужно смотреть DMV. Имейте в виду, что sqlservr.exe не сможет использовать всю оперативную память. Если на сервере 128 GB, то sqlservr.exe может использовать только 80-90% (100-110 GB), так как остальная память резервируется для операционной системы.

Имейте в виду, что процесс SQL Server’a не отдаёт оперативную память обратно в систему. Например, ваш SQL Server обычно использует 20 GB памяти, но при месячном отчете он увеличивает потребление до 100 GB, и даже когда вычисление отчета закончится и сервер будет работать в прежнем режиме, процесс SQL Server’a всё равно будет использовать 100 GB до перезагрузки службы.

Даже если вы уверены, что оперативной памяти серверу хватает, не будет лишним точно знать объём потребляемой RAM.

Узнать реальное использование RAM можно с помощью **Dynamic Management Views**. DMV это административные вьюверы (представления). С помощью DMV можно диагностировать практически любую проблему в SQL Server.

Посмотрим sys.dm\_os\_sys\_memory, для удобства используем запрос:

SELECT total\_physical\_memory\_kb / 1024 AS

[Total Physical Memory],

available\_physical\_memory\_kb / 1024 AS

[Available Physical Memory],

total\_page\_file\_kb / 1024 AS

[Total Page File (MB)],

available\_page\_file\_kb / 1024 AS

[Available Page File (MB)],

100 - ( 100 \* Cast(available\_physical\_memory\_kb AS DECIMAL(18, 3)) / Cast

(

total\_physical\_memory\_kb AS DECIMAL(18, 3)) ) AS

'Percentage Used',

system\_memory\_state\_desc AS

[Memory State]

FROM sys.dm\_os\_sys\_memory;

Рассмотрим каждый выводимый параметр:

[**Total Physical Memory**] – объём оперативной памяти доступный в операционной системе. На некоторых серверах может показывать немного больше реально установленной.

[**Available Physical Memory**] – объём оперативной памяти доступный для SQL Server, без учета уже захваченной SQL Server.

[**Total Page File (MB)]**– Объём “Сommit limit”. Commit Limit = Оперативная память + все файлы подкачки. То есть, если у вас на сервере 32 GB оперативной памяти и 16 GB файл подкачки, commit limit будет 48 GB.

[**Available Page File (MB)**] – Объём файла подкачки.

**Percentage Used** – процент занятой оперативной памяти. Такого параметра нет в самом sys.dm\_os\_sys\_memory, но он считается по формуле available\_physical\_memory\_kb / total\_physical\_memory\_kb

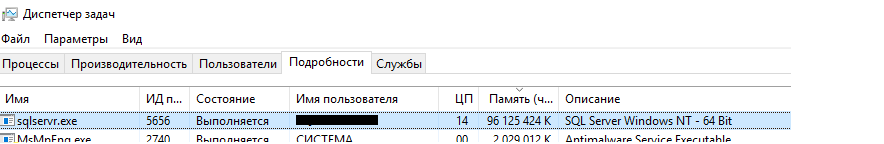
[**Memory State**] – Состояние RAM. Поле system\_memory\_state\_desc содержит в себе состояние потребления оперативной памяти в виде текста. Значение этого поля считается исходя из других двух: system\_low\_memory\_signal\_state и system\_high\_memory\_signal\_state. Вы можете выбирать их напрямую, если вам нужен Boolean/bit формат данных.

Все эти данные полезны, если вы хотите точно определить сколько ваш SQL Server потребляет RAM. Чаще всего это используют, если есть подозрения что для экземпляра выделено слишком много оперативной памяти.

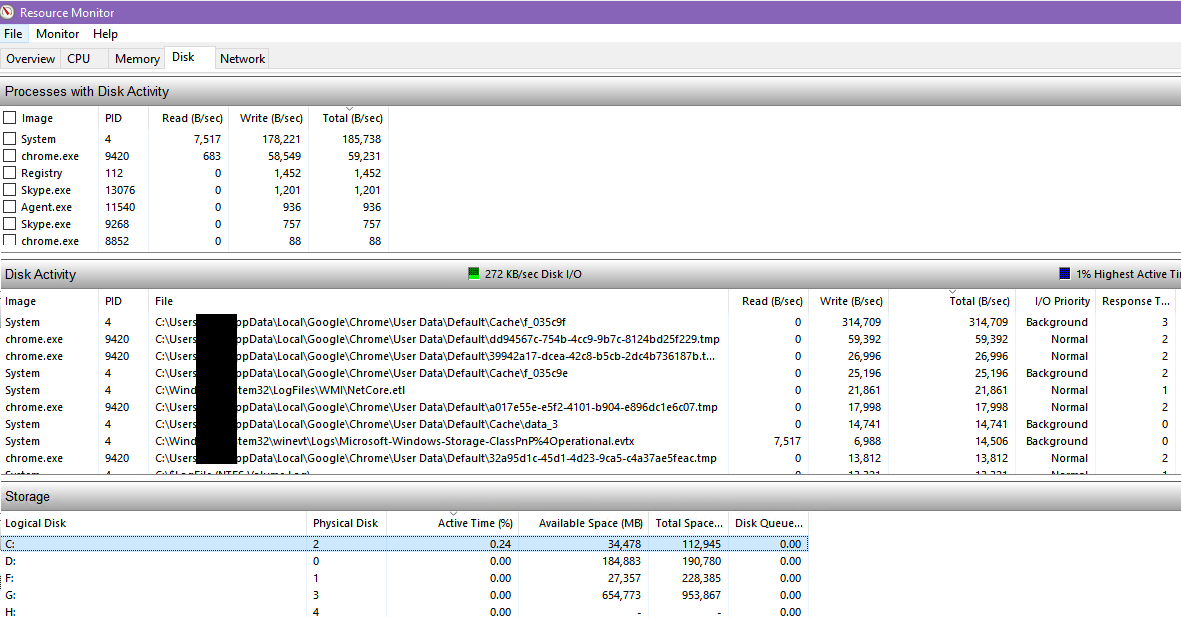
Если Вам нужно убедиться, что серверу хватает RAM, вы можете смотреть только на поля system\_low\_memory\_signal\_state, system\_high\_memory\_signal\_state и system\_memory\_state\_desc. Если system\_low\_memory\_signal\_state = 1, то серверу явно не хватает оперативной памяти.

Загрузка процессора в SQL Server

Нагрузку на процессор определить проще, так как это можно сделать в Диспетчере задач. Чтобы узнать текущую нагрузку на процессор, найдите в Диспетчере задач процесс sqlservr.exe



Анализ нагрузки на диск SQL Server

Посмотрим на загрузку дисков в операционной системе. Для этого запустите resmon.exe.

Нам нужна вкладка **Disk**. В секции **Disk Activity** отображаются файлы, к которым идёт обращение, и их скорость read/write на текущий момент. Отфильтруйте эту секцию по Total (кликните на Total). На самом верху будут файлы, которые на данный момент максимально используют диск. В случае с SQL Server это может быть полезно чтобы определить какая база больше всего нагружает диск на текущий момент.

В секции Storage отображаются все диски в системе. В этой секции нам нужны 2 параметра – Active Time и Disk Queue. Active Time в процентах отображает нагрузку на диск, то есть если вы видите на диске C:\ Active Time равный 90, это значит что ресурс чтения/записи диска на текущий момент используется на 90%. Столбец Disk Queue отображает очередь обращений к диску, и если значение очереди не равно нулю, то диск загружен на 100% и не справляется с нагрузкой. Так же если Active Time близок к 100, то диск используется практически на пределе своих возможностей по скорости.

Просмотр блокировок в SQL Server

После того как мы убедились, что серверу хватает ресурсов, можно переходить к просмотру блокировок.

Блокировки можно посмотреть через Activity Monitor в SSMS, но мы воспользуемся T-SQL, так как этот вариант более удобен и нагляден. Выполняем запрос:

SET NOCOUNT ON

GO

SELECT SPID, BLOCKED, REPLACE (REPLACE (T.TEXT, CHAR(10), ' '), CHAR (13), ' ' ) AS BATCH

INTO #T

FROM sys.sysprocesses R CROSS APPLY sys.dm\_exec\_sql\_text(R.SQL\_HANDLE) T

GO

WITH

BLOCKERS (SPID, BLOCKED, LEVEL, BATCH)

AS

(

SELECT SPID,

BLOCKED,

CAST (REPLICATE ('0', 4-LEN (CAST (SPID AS VARCHAR))) + CAST (SPID AS VARCHAR) AS VARCHAR (1000)) AS LEVEL,

BATCH

FROM #T R

WHERE (BLOCKED = 0 OR BLOCKED = SPID)

AND EXISTS (SELECT \*

FROM #T R2

WHERE R2.BLOCKED = R.SPID AND R2.BLOCKED <> R2.SPID)

UNION ALL

SELECT R.SPID,

R.BLOCKED,

CAST (BLOCKERS.LEVEL + RIGHT (CAST ((1000 + R.SPID) AS VARCHAR (100)), 4) AS VARCHAR (1000)) AS LEVEL,

R.BATCH

FROM #T AS R

INNER JOIN BLOCKERS ON R.BLOCKED = BLOCKERS.SPID

WHERE R.BLOCKED > 0 AND R.BLOCKED <> R.SPID

)

SELECT N' ' + REPLICATE (N'| ', LEN (LEVEL)/4 - 1) +

CASE WHEN (LEN(LEVEL)/4 - 1) = 0

THEN 'HEAD - '

ELSE '|------ ' END

+ CAST (SPID AS NVARCHAR (10)) + N' ' + BATCH AS BLOCKING\_TREE

FROM BLOCKERS

ORDER BY LEVEL ASC

GO

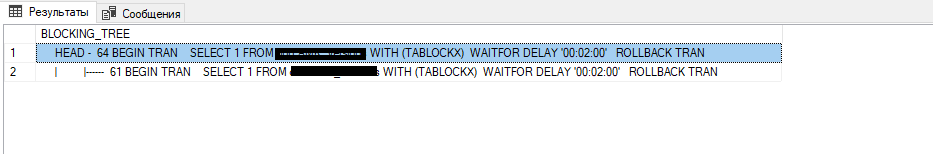
DROP TABLE #T

GO

Этот запрос возвращает список блокировок в виде дерева. Это удобно в работе, так как обычно, если возникает одна блокировка, она провоцирует за собой другие. Аналогично в Activity Monitor или в выводе sp\_who2 можно увидеть поле “Blocked By”.

Если запрос ничего не вернул, то блокировок нет.

Если запрос вернул какие-то данные, то нужно проанализировать цепочку.



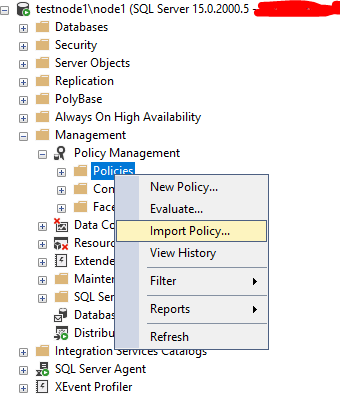
HEAD значит что этот запрос является причиной всех остальных блокировок ниже по дереву. 64 – это идентификатор процесса (SPID). После этого пишется тело запроса, который вызвал блокировку. Если у вас хватает ресурсов сервера, то скорее всего дело в самом запросе и во взаимном обращении к каким-то объектам. Для того чтобы сказать точнее, нужно анализировать конкретный запрос, который вызвал блокировку.

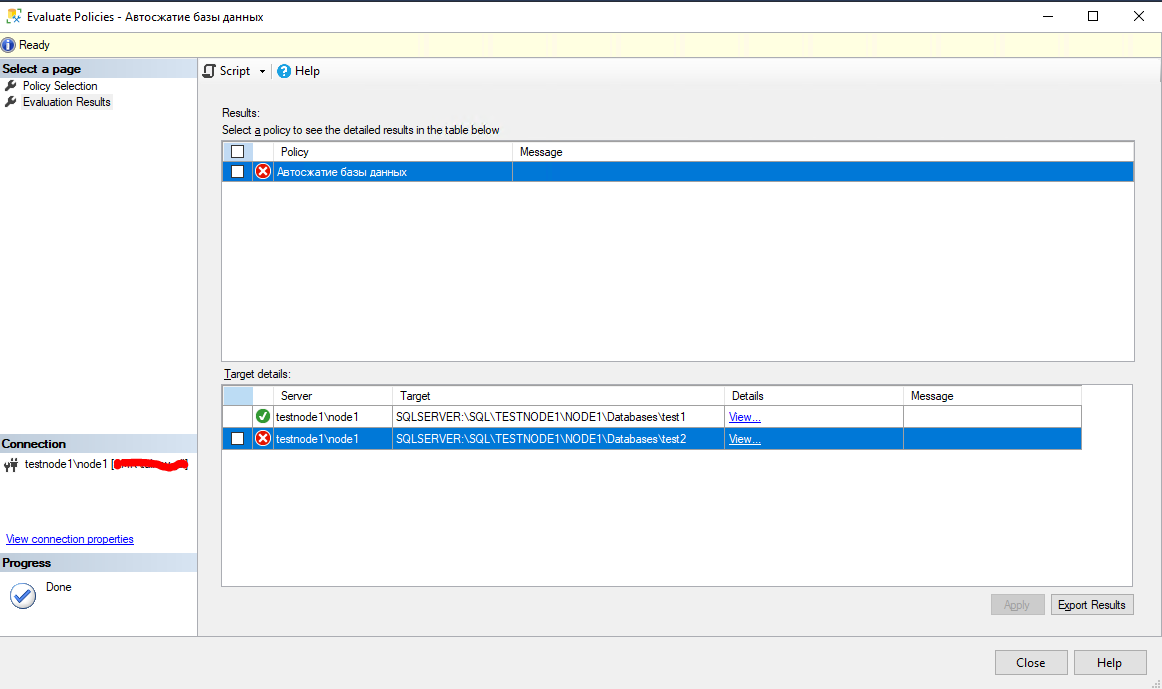
Политики SQL Server

Даже когда у вас всё работает хорошо и жалоб нет, на самом деле может быть много проблем, которые всплывут позже. Для этого в SQL Server есть политики.

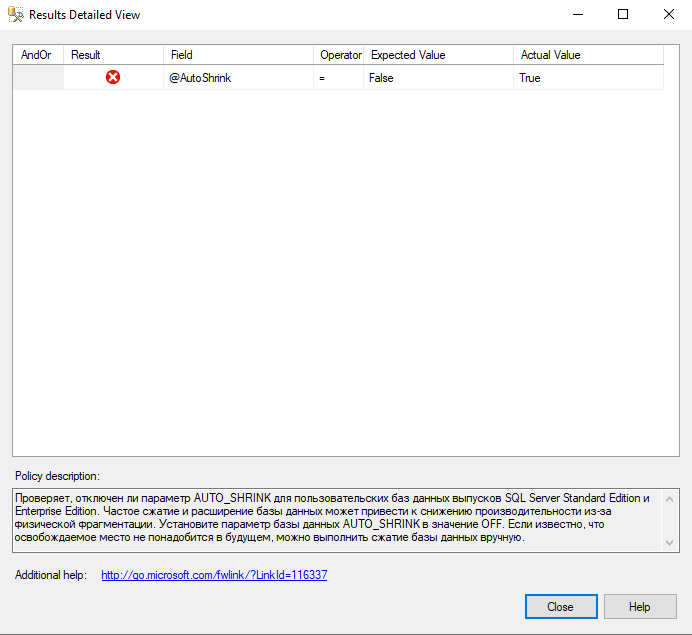
Политика в SQL Server это, грубо говоря, проверка правила на соответствие заданному значению. Например, с помощью политик вы можете убедиться, что на всех базах на сервере выключен Auto Shrink. Рассмотрим пример импорта и выполнения политики

В SSMS, подключитесь к серверу, на котором хотите выполнять политики (Management -> раздел Policy Management).



Импортируем файл Database Auto Shrink.xml. Жмём **Evaluate**

На экземпляре node1 две базы данных, test1 и test2. На test2 включен autoshrink. Посмотрим детали

.

Политика определила включенный параметр AutoShrink, в описании обычно пишется объяснения к правилам. В данном случае дается объяснение почему auto shrink лучше отключать.

Политики могут выполняться либо по расписанию, либо по требованию (разово). Результаты выполнения политики можно посмотреть в журнале политик.

При установке SQL Server нужно выбирать только используемые компоненты СУБД, и указывать настройки в соответствии с конфигурацией “железа” вашего сервера. Всегда следите чтобы серверу хватало ресурсов, и чтобы на сервере не было блокировок

Самым мощным инструментом для диагностики SQL Server является T-SQL и DMV. Так же рекомендуется построить круглосуточный мониторинг над SQL Server и над обслуживающей его инфраструктурой для обнаружения всех возможных проблем.

В этой статье описываются преимущества резервного копирования баз

В этой статье приводятся общие сведения о резервном копировании SQL Server. Конкретные действия по резервному копированию баз данных SQL Server см. в разделе Создание резервных копий.

Компонент резервного копирования и восстановления SQL Server обеспечивает необходимую защиту важных данных, хранящихся в базах данных SQL Server. Чтобы свести к минимуму риск необратимой потери данных, необходимо создавать резервные копии баз данных для сохранения вносимых изменений на регулярной основе. Хорошо спланированная стратегия резервного копирования и восстановления защищает базы от потери данных, вызванной разными сбоями. Протестируйте стратегию, выполнив восстановление набора резервных копий и вернув в исходное состояние базу данных. Так вы будете готовы эффективно реагировать на проблемы.

Помимо локального хранилища для хранения резервных копий SQL Server также поддерживает резервное копирование и восстановление из Хранилище BLOB-объектов Azure.

Резервное копирование баз данных SQL Server, выполнение тестовых процедур восстановления резервных копий в резервных копиях и хранение копий резервных копий в безопасном расположении вне сайта защищает вас от потенциально катастрофической потери данных. **Резервное копирование — единственный способ защитить данные.**

При правильном создании резервных копий баз данных можно будет восстановить данные после многих видов сбоев, включая следующие:

Сбой носителя.

ошибки пользователей (например, удаление таблицы по ошибке);

сбои оборудования (например, поврежденный дисковый накопитель или безвозвратная потеря данных на сервере);

Стихийные бедствия. С помощью резервного копирования SQL Server для Хранилище BLOB-объектов Azure можно создать резервную копию вне сайта в другом регионе, отличном от локального расположения, для использования в случае стихийных бедствий, влияющих на локальное расположение.

Кроме того, резервные копии базы данных полезны для обычных административных целей, таких как копирование базы данных с одного сервера на другой, настройка групп доступности AlwaysOn или зеркало баз данных и архивация.

Стратегии резервного копирования и восстановления

Операции резервирования и восстановления данных следует адаптировать под конкретную среду с учетом доступных ресурсов. Таким образом, для надежного использования резервного копирования и восстановления требуется стратегия резервного копирования и восстановления. Хорошо спроектированная стратегия резервного копирования и восстановления позволяет сбалансировать бизнес-требования для обеспечения максимальной доступности данных и минимальной потери данных, учитывая стоимость обслуживания и хранения резервных копий.

Стратегия резервирования и восстановления состоит из части, относящейся к резервированию, и части, относящейся к восстановлению. Часть, относящаяся к резервированию, определяет тип и частоту создания резервных копий, тип и скоростные характеристики оборудования, необходимого для их создания, способ проверки резервных копий, а также местонахождение и тип носителя резервных копий (включая и вопросы безопасности). Часть, относящаяся к восстановлению, определяет ответственного за проведение операций восстановления, методы их проведения, позволяющие удовлетворить требования пользователей по доступности базы данных и минимизации потерь данных, а также методы проверки восстановления.

Разработка эффективной стратегии резервирования и восстановления требует тщательного планирования, реализации и тестирования. Проверка необходима — у вас нет стратегии резервного копирования до тех пор, пока не были успешно восстановлены все резервные копии во всех сочетаниях, включенных в стратегию восстановления, и не была проверена физическая согласованность восстановленной базы данных. Необходимо оценить ряд факторов. Например:

Задачи организации, относящиеся к рабочей базе данных, особенно требования к доступности данных и их защите от потери или повреждения.

Свойства каждой базы данных: размер, типичное использование, характер содержимого, требования к данным и т. д.

Ограничения на ресурсы, например: оборудование, персонал, пространство для хранения носителей резервных копий, физическая безопасность этих носителей и так далее.

Рекомендации

Учетные записи, выполняющие операции резервного копирования или восстановления, не должны предоставляться больше привилегий, чем необходимо. Просмотрите сведения о резервном копировании и восстановлении для определенных сведений о разрешениях. Рекомендуется шифровать резервные копии и при возможности сжимать их.

Чтобы обеспечить безопасность, файлы резервного копирования должны иметь расширения, которые соответствуют соответствующим соглашениям:

Файлы резервной .BAK копии базы данных должны иметь расширение

Файлы резервного копирования журналов .TRN должны иметь расширение.

Использование отдельного хранилища

 Важно!

Убедитесь, что резервные копии базы данных размещаются в отдельном от файлов базы данных физическом хранилище или устройстве. Если физический диск, на котором хранятся базы данных, неисправен или завершается сбоем, восстановление зависит от возможности доступа к отдельному диску или удаленному устройству, в котором хранятся резервные копии. Помните, что можно создать несколько логических томов или разделов одного и того же физического диска. Прежде чем выбирать место хранения резервных копий, внимательно изучите разделы диска и структуры логических томов.

Выбор подходящей модели восстановления

Операции резервного копирования и восстановления выполняются в контексте модели восстановления. Модель восстановления является свойством базы данных, которое задает, как выполняется управление журналом транзакций. Таким образом, модель восстановления базы данных определяет, какие типы резервных копий и сценарии восстановления поддерживаются для базы данных и каким будет размер резервных копий журнала транзакций. Обычно база данных использует простую модель восстановления или модель полного восстановления. Модель полного восстановления может быть дополнена путем переключения на модель с неполным протоколированием перед выполнением массовых операций. Основные сведения о моделях восстановления и их влиянии на управление журналом транзакций см. в разделе Журнал транзакций (SQL Server).

Лучший выбор модели восстановления базы данных зависит от бизнес-требований. Чтобы избежать управления журналом транзакций и упростить резервное копирование и восстановление, используйте простую модель восстановления. Чтобы снизить вероятность потери результатов работы ценой увеличения административных издержек, используйте модель полного восстановления. Чтобы уменьшить влияние на размер журнала во время операций с неполным протоколированием и при этом обеспечить возможность восстановления этих операций, используйте модель восстановления с неполным протоколированием. Сведения о влиянии моделей восстановления на резервное копирование и восстановление см. в обзоре резервного копирования (SQL Server).

Создание стратегии резервного копирования

После того как выбрана модель восстановления, соответствующая бизнес-требованиям определенной базы данных, необходимо спланировать и выполнить соответствующую стратегию резервного копирования. Оптимальная стратегия зависит от многих факторов, среди которых наиболее важны следующие.

Сколько часов в день приложения имеют доступ к базе данных?

Если существует прогнозируемый внепиковый период, рекомендуется запланировать полное резервное копирование базы данных именно на этот период.

Насколько часты и вероятны изменения и обновления?

Если изменения часты, учтите следующее.

В рамках простой модели восстановления рассмотрите возможность запланировать разностное резервное копирование между полными резервными копированиями базы данных. Разностная резервная копия сохраняет только те изменения, которые были внесены с момента последнего полного резервного копирования.

В рамках модели полного восстановления следует запланировать частное резервное копирование журналов. Планирование разностного резервного копирования между полными резервными копиями сокращает время восстановления путем сокращения количества резервных копий журналов, которые необходимо восстанавливать после восстановления данных.

Касаются ли обычно изменения небольшой или же значительной части базы данных?

Для большой базы данных, в которой изменения сосредоточены в части файлов или файловых групп, частичные резервные копии и полные резервные копии файлов могут быть полезны. Дополнительные сведения см. в статьях о частичных резервных копиях (SQL Server) и полных резервных копий файлов (SQL Server).

Сколько места на диске требуется для полного резервного копирования базы данных?

За какой прошлый период компании нужны резервные копии?

Убедитесь, что установлен правильный график резервного копирования в соответствии с потребностями приложения и бизнес-требованиями. По мере устаревания резервных копий риск потери данных выше, если нет способа повторного создания всех данных до точки сбоя. Прежде чем удалять старые резервные копии из-за ограничений ресурсов хранилища, следует учесть, требуется ли возможность восстановления за этот период.

Оценка размера полной резервной копии базы данных

Перед тем как выбрать стратегию резервного копирования и восстановления, необходимо рассчитать, какой объем места на диске необходим для полной резервной копии базы данных. При выполнении операции резервного копирования данные, содержащиеся в базе данных, копируются в файл резервной копии. Резервная копия содержит только фактические данные в базе данных, а не любое неиспользованное пространство. Поэтому резервная копия обычно меньше, чем база данных. Размер полной резервной копии базы данных вы можете вычислить с помощью системной хранимой процедуры **sp\_spaceused** . Дополнительные сведения см. в разделе sp\_spaceused (Transact-SQL).

Создание расписания резервного копирования

Влияние, оказываемое осуществлением резервного копирования на выполняемые транзакции, минимально, поэтому операции резервного копирования могут выполняться одновременно с выполнением обычных операций. Резервное копирование SQL Server можно выполнять с минимальным воздействием на рабочие нагрузки.

Сведения о ограничениях параллелизма во время резервного копирования см. в разделе "Обзор резервного копирования" (SQL Server).

После принятия решения о том, какой тип резервного копирования необходим и как часто его выполнять, рекомендуется запланировать регулярное резервное копирование как часть плана обслуживания базы данных. Дополнительные сведения о планах обслуживания и об их создании для резервных копий баз данных и журналов см. в разделе Use the Maintenance Plan Wizard.

Тестирование резервных копий

Можно сказать, что стратегия восстановления отсутствует, пока резервные копии не протестированы. Очень важно полностью протестировать стратегию резервного копирования для каждой базы данных, восстанавливая копию базы данных в систему тестирования. Необходимо протестировать восстановление каждого типа резервной копии, которую планируется использовать. Также рекомендуется после восстановления резервной копии выполнить проверку согласованности базы данных с помощью инструкции DBCC CHECKDB базы данных, чтобы убедиться, что носитель резервной копии не поврежден.

Проверка стабильности и согласованности носителя

Используйте параметры проверки, предоставляемые служебными программами резервного копирования (команда BACKUP T-SQL, планы обслуживания SQL Server, программное обеспечение резервного копирования или решение и т. д.). Пример см. в разделе [RESTORE VERIFYONLY] (../t-sql/statements/restore-statements-verifyonly-transact-sql.md) Используйте дополнительные функции, например BACKUP CHECKSUM, чтобы выявить проблемы с самим носителем резервного копирования. Дополнительные сведения см. в статье Возможные ошибки носителей во время резервного копирования и восстановления (SQL Server)

Стратегия резервного копирования и восстановления документов

Рекомендуется документировать процедуры резервирования и восстановления и хранить копию этой документации в документации по задаче. Также рекомендуется вести руководство по эксплуатации для каждой базы данных. Такое руководство по эксплуатации должно указывать расположение резервных копий, имена устройств резервного копирования (если есть), время, требуемое для восстановления тестовой резервной копии.

Отслеживание хода выполнения с помощью xEvent

Операции резервного копирования и восстановления могут выполняться в течение длительного времени из-за размера базы данных и сложности выполняемых операций.

С помощью расширенных событий backup\_restore\_progress\_trace можно повысить производительность и существенно сэкономить дисковое пространство. Используйте эти события в течение короткого периода времени, проявляйте внимание и тщательно проверяйте свои изменения перед их реализацией в рабочей среде.

**Обзор технологий доступа к данным. Анализ технологий. Выбор оптимальной технологии для доступа к данным при разработке приложения.**

Технологии доступа к данным являются прослойкой между API конкретного сервера и приложением пользователя, предоставляя программисту простой унифицированный механизм работы с данными.

На сегодняшний день существует множество технологий доступа к данным, таких как BDE, OLE, ODBC, DАО, АDО, и до сих пор разрабатываются новые, более надежные, удобные в работе и более быстродействующие технологии.

Механизмы доступа к **базам данных** снижают сложность обмена информацией с базами, однако интерпретация результатов их работы также достаточно трудоемка. Поэтому реализованы наборы компонентов, предназначенные для взаимодействия с механизмами обмена.

Таким образом, можно выделить несколько субъектов, участвующих в движении информации между **базой данных** и приложением (например, пользовательским интерфейсом):

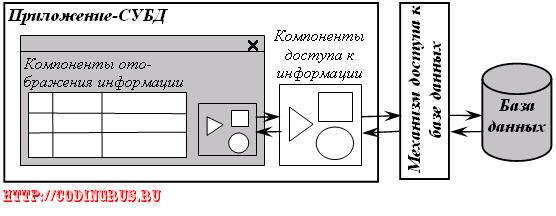
1) интерфейсная часть приложения или его программная часть, манипулирующая информацией, хранимой в базе данных.;

2) компоненты, обеспечивающие связь приложения с механизмом доступа к базе данных;

3) механизм доступа к базе данных;

4) база данных.

На рис. 1 представлена схема движения информации между приложением и базой данных.



**Рис. 1. Движение информации от базы данных к приложению**

при разработчике приложения-СУБД программист работает с наборами компонентов, предназначенных для обмена информацией с базами данных и ее отображения. В зависимости от выбранного механизма доступа к базе данных некоторые наборы компонентов могут не использоваться, однако все они, вне зависимости от особенностей используемой базы данных и механизма доступа к ней, имеют схожие свойства и методы.

**ODBC (Open Database Connectivity – открытый доступ к базам данных) – разработанный компанией Microsoft универсальный** интерфейс программирования приложений **для доступа к базам данных**.

Основной целью разработки протокола ODBC считается стандартизация механизмов взаимодействия с различными СУБД. Основная проблема, связанная с разработкой приложений, взаимодействующих с базами данных на основе специальных SQL API, состояла в том, что каждая СУБД имела собственный программный интерфейс доступа, каждый из них имел свои особенности и функционировал не совсем так, как другие. В связи с этим разработка приложения существенно зависела от используемой СУБД. Компания Microsoft сделала важный шаг для решения этой проблемы. Основная идея заключалась в разработке универсального интерфейса на уровне семейства операционных систем Windows, который мог бы быть поддержан в разных СУБД.

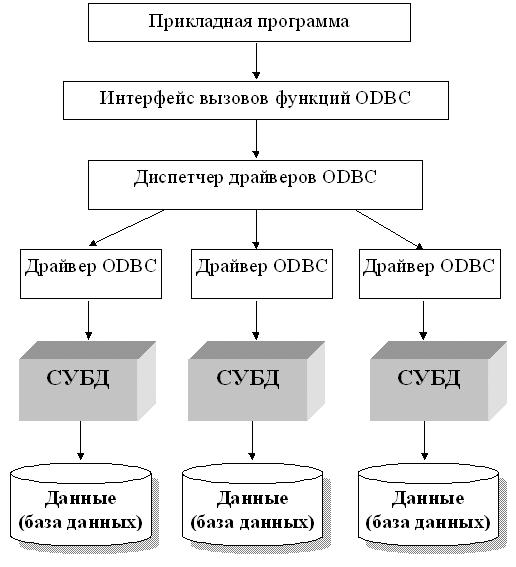
Рассмотрим кратко структуру программного обеспечения ODBC:

**интерфейс вызовов функций ODBC**: это так называемый верхний уровень ODBC, содержащий API, который и используется непосредственно приложениями. Данный API реализован в виде библиотеки динамической компоновки Dll и входит в состав операционной системы Windows;

**драйверы ODBC**: это так называемый нижний уровень ODBC, содержащий набор драйверов для СУБД, поддерживающих протокол ODBC. В рамках технологии для каждой СУБД может быть разработан соответствующий ODBC-драйвер, который будет являться промежуточным звеном между прикладной программой и СУБД, транслируя вызовы функций СУБД в вызовы внутренних специализированных функций СУБД. Таким образом решается проблема стандартизации. Для многих современных СУБД существуют специализированные драйверы ODBC, отдельно устанавливаемые в операционную систему;

**диспетчер драйверов ODBC**: данный программный механизм представляет средний уровень ODBC, управляя процессом загрузки необходимых драйверов.

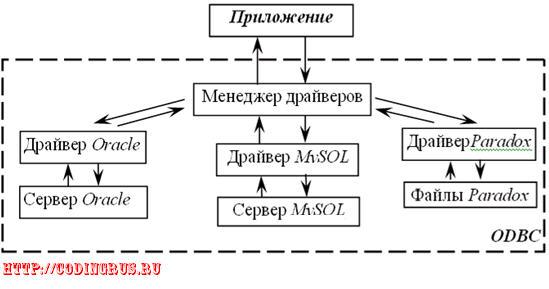
Схема выполнения программы с использованием протокола ODBC для доступа к данным приводится на рис.2.



**Рис. 2.**  Схема выполнения программы с использованием протокола ODBC для доступа к данным

Операционная система Windows имеет в своем составе несколько механизмов доступа к базам данных: *ODBC*, *OLE DB* и *ADO*.

**Технология ODBC** (от англ. *Open Database Connectivity* – открытый механизм доступа к базам данных**1** ) - это компонент операционной системы *Windows*, предназначенный для унификации доступа к информации, хранящейся в **базах данных** различных видов. *ODBC* состоит из набора драйверов, осуществляющих операции обмена с определенными **базами данных**, и менеджера драйверов, осуществляющего передачу запросов от приложения к драйверу и передачу информации от драйвера к приложению (рис. 3).

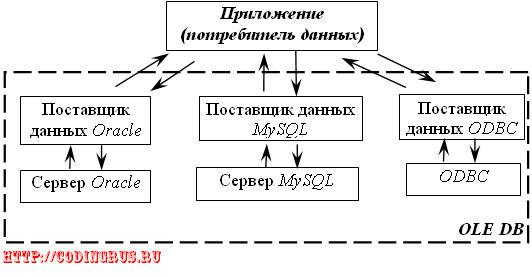


**Рис. 3. Движение информации между приложением и БД при использовании ODBC**

Для получения и изменения данных используется язык запросов **SQL**, вне зависимости от того, поддерживается ли он базой данных, к которой обращается приложение. Если база данных не поддерживает язык **SQL**, то доступ к ней не отличается от доступа к **БД**, поддерживающим **SQL**. В этом и заключается унификация доступа к базам данных системой **ODBC** – приложение указывает название драйвера, который должен использоваться для подключения к базе данных, и передает запрос, в котором описан состав требуемой информации. Далее механизм **ODBC** выполняет все необходимые операции по получению информации, скрывая от приложения специфику работы с конкретной базой данных. Доступ приложений к **ODBC** осуществляется через **API**-функции, реализованные в динамических библиотеках.

**Технология OLE DB** (от англ. *Object Linking and Embedding DataBase* – механизм доступа к базам данных с помощью технологии *OLE*) представляет собой набор **СОМ**-интерфейсов, обеспечивающих универсальный доступ приложений к базам данных и другим хранилищам информации (в том числе и нереляционным - файловым системам, системам электронной почты и др.), в которых результатом за-просов является информация, представленная в табличной форме.

Характерной особенностью механизма *OLE DB* является то, что данные, возвращаемые *OLE DB* приложению, представляют собой не просто массив информации, а *СОМ*-объект, обладающий, помимо самой информации, методами управления этой информацией (например, фильтрация и сортировка). Логическая схема доступа к данным с помощью *OLE DB* представлена на рис. 4.

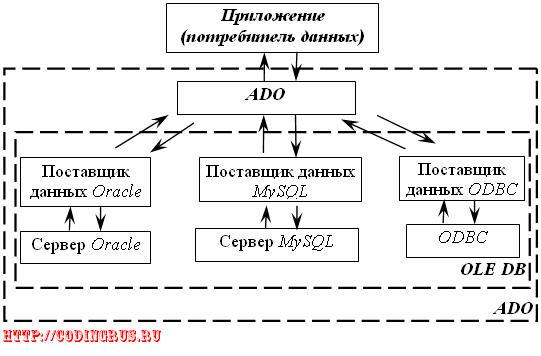


**Рис. 4. Движение информации между приложением и БД при использовании OLE DB**

Приложение, называемое потребителем данных, обращается к поставщику данных, который перенаправляет запрос от приложения к базе данных, либо обрабатывает запрос самостоятельно. Таким образом, поставщик данных аналогичен драйверам *ODBC* и *BDE*. Механизм *OLE DB* имеет доступ к *ODBC*, поддерживая, таким образом, большое количество драйверов, реализованных для *ODBC*.

Недостатком *OLE DB* (и соответственно технологии *ADO*, построенной на *OLE DB*) является более низкое быстродействие по сравнению с механизмами *ODBC* и *BDE*. Это связано с использованием **СОМ**-объектов, применение которых нагружает операционную систему. Однако работа механизмов связи с базой данных при использовании *OLE DB* обычно занимает меньшее время по сравнению с работой самой базы данных по выдаче запроса.

**Технология ADO**(от англ. *ActiveX Database Objects* – механизм доступа к базам данных через объекты ActiveX) – технология доступа к данным, разработанная *Microsoft*, является надстройкой над механизмом доступа *OLE DB* и предназначена для унификации работы с поставщиками данных *OLE DB*. Обеспечивает удобный и надежный доступ к данным, хотя и несколько более медленный, чем в технологиях *BDE* и *dbExpress*. Хорошо подходит для работы с системами управ-ления базами данных (СУБД) от фирмы *Microsoft* (*MS Access, MS SQL Server*), поскольку не требует дополнительных компонентов (библиотек, драйверов), так как они уже есть на компьютере пользователя. Схема доступа к информации с помощью *ADO* представлена на рис. 5.



**Рис. 5. Движение информации между приложением и БД при использовании ADO**

Механизм **ADO** предоставляет несколько основных **СОМ**-объектов, используемых для получения и управления информацией (имеются дополнительные **СОМ**-объекты, расширяющие функциональность *ADO*):

• Connection для управления соединением с базой данных и передачи запросов поставщику данных;

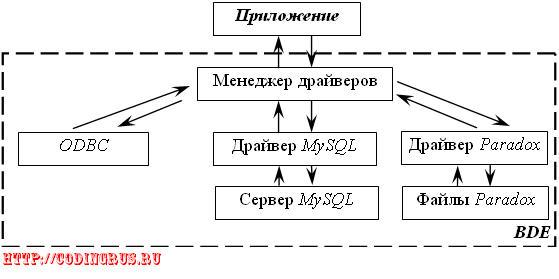
• Command для управления информацией о запросе к базе данных или команде;

• Recordset, содержащий таблицу, которая является результатом запроса к базе данных;

• Field, содержащий описание поля в таблице, возвращенной поставщиком данных. Список всех полей таблицы содержится в подобъекте Fields объекта RecordSet;

• Error, содержащий расширенную информацию об ошибке, о которой сообщил поставщик данных. Если ошибок несколько, доступ к ним можно получить с помощью объекта Errors.

**BDE**(англ. *Borland Database Engine* – механизм доступа к базам данных фирмы *Borland*) – базовая технология доступа к БД от фирмы *Borland*, является аналогом *ODBC* и имеет схожую с ней архитектуру. Этот механизм позволяет получать доступ к реляционным БД с помощью специальных *BDE* драйверов или через *ODBC* драйверы. Исторически является первой технологией доступа к БД в средах *Borland*. Существенным недостатком использования этой технологии является достаточно трудоемкий процесс развертывания программы работы с базой данных: помимо самой программы на компьютере пользователя необходимо установить *BDE* и выполнить его настройку, а также отсутствие поддержки со стороны фирмы-разработчика.



**Рис. 6. Движение информации между приложением и БД при использовании BDE**

**dbExpress** – более новая и совершенная технология доступа к удаленным БД от фирмы *Borland*. Схема работы этого механизма аналогична схеме работы *BDE*, но при использовании *dbExpress* не происходит обращения к *ODBC*. При использовании этого механизма нет необходимости развертывания *dbExpress* вместе с приложением: достаточно установить только **dll**-библиотеку либо прилинковать драйвер к исполняемому файлу. Технология *dbExpress* по сравнению с *BDE* обеспечивает более быстрый и удобный доступ к данным.

**InterBase**– специализированная технология доступа к серверу БД *Borland InterBase*. При работе с этой СУБД обеспечивает наивысшую производительность по сравнению с другими технологиями.

Принципы работы с БД с использованием любой из вышеперечисленных технологий практически одинаковы.

**Протокол JDBC**

JDBC (Java Database Connectivity) представляет собой API для выполнения SQL-запросов к базам данных из программ, написанных на языке Java.

С развитием глобальных сетей, в частности Интернета, и всех сопутствующих технологий стали появляться новые языки, специально предназначенные для работы в новых условиях. Одним из таких языков является язык программирования Java. В настоящее время Интернет-приложения занимают существенное место на рынке, работая в рамках 2-, 3- и многозвенной архитектуры. При этом значение языка Java как средства создания приложений, работающих с базами данных, существенно возрастает. Именно это и явилось одной из основных причин разработки нового программного интерфейса – JDBC. Первоначально интерфейс JDBC был разработан компанией Sun Microsystems, в настоящий момент этот API поддерживается всеми ведущими коммерческими СУБД.

**Интерфейс среды для разработки пользовательского приложения. Описание визуальных компонентов и компонентов доступа к данным.**

GUI в переводе — это графический интерфейс пользователя. Все эти кнопки, иконки, окна, значки, вкладки, которые вы видите на своем компьютере и телефоне…

Когда вы нажимаете на кнопки на экране, выполняется большое количество кода, и именно он связывает каждое нажатие с тем действием, которое затем совершает компьютер.

Что такое графический пользовательский интерфейс (GUI)

До появления GUI пользователям приходилось писать команды в командной строке. Тот, кто хоть раз там работал, понимает, насколько это неудобно и сложно. Представьте, например, на секунду, что вы копируете кучу файлов из одной папки в другую, без перетаскивания, а вручную указывая десятки и сотни команд.

Интерфейс — это графическая оболочка программы или приложения.

GUI — это интерфейс, с помощью которого пользователь взаимодействует с визуальными элементами программы.

Элементы GUI — пиктограммы, значки, окна, кнопки. Эти и другие компоненты выводятся в рабочую область программы, затем пользователь взаимодействует с ними используя указатель, клавиатуру или сенсорный экран.

Графический интерфейс пользователя произвел революцию в компьютерах, сделав их более интуитивными и удобными. Сегодня графические интерфейсы есть у любой программы или приложения, независимо от устройства или операционной системы.

Как создаются GUI

Графические интерфейсы разрабатываются в зависимости от используемых устройств ввода (тачскрин, мышь, клавиатура). Например, для сенсорного экрана (смартфоны и мобильные устройства высокого класса) требуются более крупные значки — вспомните приложения на вашем телефоне или планшете:

А вот на десктопе вы можете без труда взаимодействовать даже с мелкими иконками и другими элементами GUI — при помощи мыши.

**В чем отличия от UI**

UI или как его еще называют «пользовательский интерфейс» — это взаимодействие между пользователем и конкретным устройством. Пользователь вводит данные, а затем видит результат в выводе. Разница между UI и GUI в том, что пользовательский интерфейс не обязательно должен быть графическим: бывают интерфейсы скринридеров, интерфейсы командной строки и многие другие.

Таким образом, GUI — это подвид UI.

Элементы GUI: какие бывают

GUI использует, в основном, визуальные элементы. Эти элементы и определяют визуальную часть GUI или то, как мы видим графическую оболочку программы. Ниже мы разберём самые частые элементы графического пользовательского интерфейса.

Окно

Это главный элемент, который отображает информацию на экране. Окном легко манипулировать: его можно открыть или закрыть кликом по иконке. Более того, его можно переместить в любую область путем перетаскивания. В многозадачной среде несколько окон могут быть открыты одновременно, и все они выполняют различные задачи.

В GUI предусмотрено несколько типов окон, например, окно-контейнер, окно браузера, окно текстового терминала, дочернее окно, окно сообщения и так далее.

Меню

Меню содержит список вариантов и позволяет пользователю выбрать один из них. Строка меню отображается горизонтально по всему экрану, например, как выпадающее меню. Также существует контекстное меню: оно выводится при выполнении конкретного действия:

Иконки

Это миниатюра или маленькое изображение. Обычно иконка связана с конкретным функционалом, например, позволяет быстро запустить программу, открыть видео или изображение.

Элементы управления: виджеты

Виджет — это элемент интерфейса и примитив GUI, выполняющий какое-либо действие или действия. Виджет всегда имеет конкретное функциональное назначение. Например, он отображает календарь и погоду:

Виджетом также может быть и часть интерфейса, в которой собрано сразу несколько графических элементов для выполнения какой-либо задачи.

В виджет можно встроить все что угодно: музыкальный плеер, ящик электронной почты, стоимость акций, курс валюты или внушительную биржевую сводку.

Главное отличие виджета от других элементов в том, что он всегда на виду: его не нужно вызывать на экран какой-то кнопкой или другим способом.

Вкладка

Это небольшой выступ с текстом, который используется для переключения между несколькими экранами или приложениями в режиме одного окна. Ближайшая аналогия — закладка в бумажной книге.

Также к элементам GUI можно отнести: формы, поля ввода (текстовые поля, холсты), кнопки (CTA, значки, плавающие кнопки) и так далее.

Компоненты доступа к данным являются невизуальными компонентами. Таблицы БД располагаются на диске и являются физическими объектами. Для операций с данными, содержащимися в таблицах, используются наборы данных. В терминах системы Delphi набор данных представляет собой совокупность записей, взятых из одной или нескольких таблиц БД. Записи, входящие в набор данных, отбираются по определенным правилам, при этом в частных случаях набор данных может включать в себя все записи из связанной с ним таблицы или не содержать ни одной записи. Набор данных является логической таблицей, с которой можно работать при выполнении приложения. Взаимодействие таблицы и набора данных напоминает взаимодействие физического файла и файловой переменной12.

В Delphi для работы с наборами данных служат компоненты DBTable и ADOTable, DBQuery и ADOQuery, DataAccess, DataControl, DecisionQuery и StoredProc .

Компонент StoredProc используется для вызова хранимых процедур при организации взаимодействия с удаленными БД, а компонент UpDateSQL обеспечивает работу с кэшированными изменениями в записях. Компонент DecisionQuery применяется при построении систем принятия решений. Наиболее универсальными и, соответственно, часто используемыми являются компоненты Table и Query, задающие наборы данных.

Базовые возможности доступа к БД обеспечивает класс DataSet, представляющий наборы данных в виде совокупности строк и столбцов (записей и полей). Этот класс предоставляет основные средства навигации и редактирования наборов данных.

Компонент DataSet предназначен для представления набора данных из хранилища данных ADO. Он прост в использовании, имея только несколько собственных свойств и методов.

Это единственный компонент ADO, инкапсулирующий набор данных, для которого опубликованы свойства, позволяющие управлять командой ADO. В результате компонент представляет собой гибкий инструмент, который позволяет (в зависимости от типа команды и ее текста) получать данные из таблиц, запросов SQL, хранимых процедур, файлов и т. д.

Компоненты Table и Query являются производными от класса ADODataSet потомка класса DataSet. Они демонстрируют схожие с базовыми классами характеристики и поведение, но каждый них имеет и свои особенности.

Компонент АDOTаblе обеспечивает использование в приложениях Delphi таблиц БД, подключенных через провайдеры OLE DB. По своим функциональным возможностям и применению он подобен стандартному табличному компоненту. В основе компонента лежит использование команды ADO, но ее свойства настроены заранее и изменению не подлежат.

Другие свойства и методы компонента обеспечивают применение индексов. Так как не все провайдеры ADO обеспечивают прямое использование таблиц БД, то для доступа к ним может понадобиться запрос SQL.

Компонент **Table представляет собой набор данных, который в некоторый момент времени может быть связан только с одной таблицей БД. Этот набор данных формируется на базе навигационного способа доступа к данным, поэтому компонент Table рекомендуется использовать для локальных БД, таких как dBase и Paradox. При работе с удаленными БД следует использовать компонент Query. Связь между таблицей и компонентом Table устанавливается через его свойства TableName, которое задает имя таблицы.**

**При задании значения свойства TableName указывается имя файла и расширение имя файла. На этапе разработки приложения имена всех таблиц доступны в раскрывающемся списке Инспектора объектов. В этот список попадают таблицы, файлы которых расположены в каталоге, указанном в свойстве TableName**13**.**

**Компонент ADOQuery обеспечивает применение запросов SQL при работе с данными через ADO. По своей функциональности он подобен стандартному компоненту запроса.**

**Компонент Query представляет собой набор данных, записи которого формируются в результате выполнения SQL-запроса и основаны на реляционном способе доступа к данным. При работе с удаленными БД рекомендуется использовать набор данных Query. При работе с удаленными базами данных следует обращаться к средствам языка SQL. Это относится и к таким операциям, как перемещение по набору данных или вставка в него записей. Если же для компонента Query используются методы типа Next и Insert, то вместо реляционного способа доступа к данным будет применён навигационный. В результате набора данных Query будет мало чем отличаться от набора данных Table. В отличие от компонента Table. Набор данных Query может включать в себя записи более чем одной таблицы БД. Текст запроса, на основании которого в набор данных отбираются записи, содержится в свойстве SQL типа Strings. Запрос включает в себя команды на языке SQL и выполняется при открытии набора данных. Запрос SQL иногда называется SQL-программой. SQL-запрос можно формировать и изменять динамически, внося изменения в его текст непосредственно при выполнении приложения.**

Компоненты для работы с данными расположенные на странице DataControls палитры компонентов и предназначены для построения интерфейсной части приложения. Они используются для навигации по набору данных, а также для отображения и редактирования записей.

Одни визуальные компоненты для работы с данными предназначены для выполнения операций с полями отдельной записи, они отображают и позволяют редактировать значение поля текущей записи. К таким компонентам относятся, например, однострочный редактор Edit и графический обзор Image.

Другие компоненты служат для отображения и редактирования сразу нескольких записей. Примерами таких компонентов являются сетки DBGrid и DBCtrlGrid, выводящие записи набора данных в табличном виде. Визуальные компоненты для работы с данными похожи на соответствующие компоненты страниц Standard и Additional и отличаются, в основном, тем, что ориентированны на работу с БД и имеют дополнительные свойства DataSource и Datafield. Первое из них указывает на источник данных, а второе - на поле набора данных, с которым связан визуальный компонент. Например, Edit отображает строковое значение, позволяя пользователю изменять его.

Для всех визуальных компонентов, предназначенных для отображения и редактирования значений полей, при изменении пользователем их содержимого набор данных автоматически переводится в режим редактирования. Произведённые с помощью этих компонентов изменения автоматически сохраняются в связанных с ними полях при наступлении определённых событий.

При программном изменении содержимого эти визуальных компонентов набор данных автоматически в режим редактирования не переводится. Для этой цели в коде должен предварительно вызываться метод Edit набора. Чтобы сохранить изменения в поле (полях) текущей записи, мы должны также предусмотреть соответствующие действия, например, вызов метода Post или переход к другой записи набора данных.

В библиотеке визуальных компонентов для всех компонентов, в том числе и предназначенных для работы с данными, базовым является классу Control. Он обеспечивает основные функциональные атрибуты такие, как положение и размеры элемента, его заголовок, цвет и другие параметры. Класс Control включает в себя общие для визуальных компонентов свойства, события и методы. В целом визуальные компоненты можно разделить на две группы: оконные и неоконные.

Оконный элемент управления представляет собой специализированное окно, предназначенное для решения конкретной задачи. К таким элементам относятся, например, поля редактирования, командные кнопки, полосы прокрутки.

Такие компоненты, как Edit, DBEdit, Memo или DBMemo при получении фокуса ввода отображают в своей области курсор редактирования. Компоненты, не связанные с редактированием информации, получение фокуса ввода обычно отображают с помощью с помощью пунктирного черного прямоугольника.

К неоконным элементам управления базовым является класс GraphiControl, производимый непосредственно от класса Control. Неоконные элементы управления на могут получать фокус ввода. Их достоинством является менее расходования ресурсов.

Свойства позволяют управлять внешним видом и поведением компонентов при проектировании и при выполнении приложения. Обычно установка значений большинства свойств компонентов выполняется на этапе проектирования с помощью инспектора объектов.

Свойство Name указывает на имя компонента, которое используется для управления компонентов во время выполнения приложения. Каждый новый компонент, помещаемый на форму, получает имя по умолчанию, автоматически образуемое путем добавления к названию компонента его номера в порядке помещения на форму. На этапе разработки приложения мы можем изменять имя компонента на более осмысленное и соответствующее назначению компонента.

Свойство Aling определяет способ выравнивания компонента на самой форме, на которой оно находится. Выравнивание используется в случае, когда требуется, чтобы какой-либо интерфейсный элемент занимал определённое положение.

Свойство Caption содержит строку для надписи заголовка компонента. Отдельные символы в заголовке могут быть подчеркнуты, они означают комбинации клавиш быстрого доступа.

Свойство Color определяет цвет фона. Часто удобно задавать цвета с помощью констант. Отображаемые цвет зависит от параметров видеокарты и монитора, в первую очередь от установленного цветного разрешения.

Визуальные компоненты способны генерировать и обрабатывать достаточно большое число событий различных видов. К наиболее общим группам событий можно отнести следующие действия:

– выбор управляющего элемента;

– перемещение указателя мыши;

– нажатие клавиш клавиатуры;

– получение и потеря управляющим элементом фокуса ввода;

– перемещение объектов методом drag-and-drop.

Существуют и более сложные события, требующие передачи дополнительных параметров, например событие, связанное с перемещением указателя мыши, передаёт координаты указателя.

**Способы внедрения PHP-сценариев. Синтаксис PHP. Особенности передачи данных формы PHP-сценарию. Создание пользовательских функций. Передача параметров функции.**

Способы внедрения PHP-кода на HTML-страницу

Для того, чтобы **внедрить PHP-код в HTML-документ**, нужно поместить его между символами (теги PHP) <?php и ?>;

<?php

// PHP-код

?>

PHP-код в HTML-документ можно также внедрить и с помощью других тегов:

<?

// PHP-код

?>

<%

// PHP-код

%>

<script language="php">

// PHP-код

<script>

Но их использовать не рекомендуется.

Файл настроек PHP интерпретатора — php.ini позволяет включать или отключать способы **внедрения PHP-кода** в HTML-документ. Cпособы описанные ниже, доступны всегда, вне зависимости от настроек файла php.ini

<script language="php">

// PHP-код

<script>

Наиболее рекомендуемый способ внедрения PHP-кода в HTML-страницу, соответствует XML-стандарту:

<?php

// PHP-код

?>

Если код содержит только PHP-код, некоторые специалисты рекомендуют опускать тег, чтобы предотвратить случайное отображение пробелов. Однако это хорошая идея лишь в том случае, если код содержит только PHP.

Разделение инструкций в PHP

В PHP инструкции завершаются с помощью точки с запятой (так же, как и во многих других языках программирования).

В последней строке кода точку с запятой ставить необязательно, так как закрывающий тег php подразумевает конец кода.

Обратите внимание на то, что вывод из обеих строк кода уложился в одну строку. Это произошло потому, что мы не добавили никакого элемента HTML для создания разрыва между двумя строками. Если же просто нажать клавишу Enter, этого будет недостаточно.

Чувствительность PHP к регистру

Такие группы, как класс, управляющие структуры (например, операторы if, if else ) и имена функций не чувствительны к регистру.

Однако переменные чувствительны к регистру.

<?php  
$a = "I am not capital A";  
$A = "I am not lower case a";  
?>

В приведенном выше примере $a и $A — это не одно и то же.

Нечувствительность PHP к пробелам

Когда вы пишете код, не имеет значения, есть ли пробелы между инструкциями. Это никак не влияет на результат.

**Пример:**

<?php  
Echo "This is code";Echo "I have a pressed enter and space a few times but PHP doesn’t care";?>

Правила комментирования в PHP

Просто добавьте # перед фрагментом, к которому вы хотите добавить комментарий.

Вы также можете комментировать несколько строк, окружив весь код символом /\* \*/.

**Условные операторы**

Часто при написании кода необходимо выполнить разный код в зависимости от условий. Для этого случая и нужны условные операторы.

В PHP используются следующие условные операторы:

**if:** выполнить определенный код, если некоторое условие истинно;

**else:** если определенное условие(я) не истинно(ы), выполнить этот код;

**if else:** если вышеуказанное условие(я) не истинно(ы), выполнить этот код;

**switch:** в зависимости от значения переменной выполнить какой-то конкретный код.

Оператор if

Оператор if используется для принятия решений.

**Синтаксис**

if(condition)  
{  
 //Код для выполнения (если условие истинно)  
}

В приведенном выше примере внутри скобки находится условие, которое нужно проверить на истинность; если условие истинно, следует выполнить код внутри этих скобок {}.

Обратите внимание на то, используете ли вы оператор == корректно. Помните: если вы хотите проверить, равно ли что-то чему-то, вы должны использовать знак ==; одинарный знак равенства = означает в PHP присвоение.

Внутри фигурных скобок вы можете написать столько кода, сколько захотите. Здесь нет ограничений на количество выполняемого внутри скобок кода.

Оператор Else

Задача оператора else состоит в том, чтобы выполнять код, когда ни одно из условий if не является истинным.

Например:

<?php  
if()  
{}  
else  
{}  
?>

Оператор Else if

Итак, мы узнали назначение и способ действия операторов if и else.

Однако у нас есть проблема: если вы используете только оператор if else, перед вами всего лишь два варианта развития событий:

либо вы выполняете условие, и тогда происходит что-то одно;

либо вы не выполняете условие, и происходит что-то другое.

Следует отметить один маленький момент: нет никакой разницы между написанием else if или elseif: и то, и другое означает абсолютно одно и то же.

Разница между If и else If

Почему же иногда нужно использовать else if, а не просто if?

Эти два оператора не идентичны, и если сделать неправильный выбор, это может сказаться на работе программы.

Если вы используете else if, то при совпадении одного из условий не проверяются другие условия и работа оператора прекращается.

Пример:

<?php  
$a=22;  
if($a==22)  
{  
 echo "the variable is equal to 22";  
 echo "<br>"  
}  
else if($a%2==0)  
{  
 echo "the variable is an even number";  
}  
else  
{  
 echo "the variable is neither equal to 22 or an event number";  
}  
?>

В приведенном выше примере мы установили $a равным 22. Но 22 — это четное число, поэтому оно также соответствует второму условию. Однако программа не будет продолжать проверку дополнительных условий в цикле.

Другое дело, если бы мы написали код следующим образом:

<?php

$a=22;

if($a==22)

{

echo "the variable is equal to 22";

echo "<br>"

}

if($a%2==0)

{

echo "the variable is an even number";

}

else

{

echo "the variable is neither equal to 22 or an event number";

}

?>

Здесь два дополнительных оператора if означают, что будет проверено второе условие — является ли $a четным или нет. Следовательно, второй блок кода будет выполнен, в отличие от первого примера.

Формы — это часть языка HTML. Формы нужны для передачи данных от клиента на сервер. Чаще всего формы используются для регистрации пользователей, заполнения анкет, оформления заказа в интернет магазине, и так далее.

Через формы можно отправлять как простую текстовую информацию, так и файлы.

Большую часть времени программирования на PHP вы будете так или иначе работать с формами и данными из них.

HTML описывает то, из каких элементов состоит форма, и как она выглядит. Но без принимающей стороны, то есть сервера, который принимает эти данные и обрабатывает их нужным образом, создавать формы нет никакого смысла.

PHP содержит множество средств для работы с формами. Это позволяет очень просто решать типичные задачи, которые часто возникают в веб-программировании:

Регистрация и аутентификация пользователя;

Отправка комментариев на форумах и социальных сетях;

Оформление заказов.

Практически любой современный сайт содержит как минимум несколько разных HTML-форм.

**Отправка формы**

Рассмотрим один типичный пример — форма обратной связи. Для связи пользователей с авторами сайта, как правило, используются формы обратной связи, где человек указывает имя, почту для обратной связи и текст своего сообщения.

Такая форма в HTML может выглядеть следующим образом:

<form name="feedback" method="POST" action="form.php">

<label>Ваше имя: <input type="text" name="name"></label>

<label>Ваш email: <input type="text" name="email"></label>

<label>Сообщение: <textarea name="message"></textarea></label>

<input type="submit" name="send" value="Отправить">

</form>

Это очень простая форма, состоящая из трёх полей и одной кнопки отправки.

Почти весь приведённый код описывает внешний вид и содержание формы, но следует обратить внимание на два атрибута тега <form>, которые нужны для указания на способ обработки данных:

method — этот атрибут используется для определения метода HTTP, который будет использован для передачи данных на сервер. Вы уже знакомы с HTTP-методом GET, предписывающим серверу просто вернуть определённый документ.

Метод POST сообщает о намерении передать на сервер некоторую информацию, что, впрочем, не отменяет последующее получение контента.

action — содержит адрес PHP-скрипта, который должен обработать эту форму.

После нажатия на кнопку «отправить», браузер выполняет POST запрос со введёнными данными на адрес, указанный в атрибуте action.

**Обработка формы**

После отправки формы управление передаётся PHP-скрипту, который должен получить переданные данные, выполнить с ними какие-либо действия (например, сохранить в базе данных) и показать результат.

Результатом может быть какое-нибудь сообщение об успешном завершении операции, например, «ваши данные успешно отправлены».

Поэтому требуется в первую очередь научиться получать данные из формы в сценарии.

В PHP это делается легко — все данные из формы находятся в глобальном ассоциативном массиве $\_POST. Этот массив всегда будет неявно присутствовать в сценарии, если он был загружен по методу POST.

Каждое поле из формы будет находиться в массиве, где ключом будет значение атрибута name, а значением содержимое поля. Например, чтобы вывести из формы всю информацию на экран, можно написать такой сценарий:

<?php

if (isset($\_POST)) {

print("Имя: " . $\_POST['name']);

print("<br>Email: " . $\_POST['email']);

print("<br>Сообщение: " . $\_POST['message']);

}

?>

Функция isset служит для определения, существует ли переданная ей переменная. Так мы проверяем, что сценарий загружен методом POST, то есть была отправлена форма.

Как правило, после обработки формы в PHP, сценарий должен переадресовать пользователя на другую страницу. Это связано с тем, что если форма была отправлена через метод POST, то после обновления страницы данные будут отправлены ещё раз, а это, в большинстве случаев, нежелательное поведение.

**Отправка файлов**

Кроме текстовой информации, существует возможность отправлять на сервер файлы любых типов. Пример формы для загрузки файла:

<form name="file\_upload" method="POST" action="form.php" enctype="multipart/form-data">

<label>Ваш аватар: <input type="file" name="avatar"></label>

<input type="submit" name="send" value="Отправить файл">

</form>

Тут есть два важных отличия от первого примера:

Добавился новый атрибут enctype, который всегда должен иметь значение multipart/form-data. Если его не будет, то файл не отправится.

Сам файл загружается при помощи поля с типом file.

В PHP загруженный файл будет доступен в другом специальном массиве — $\_FILES.

<?php

if (isset($\_FILES['avatar'])) {

$file = $\_FILES['avatar'];

print("Загружен файл с именем " . $file['name'] . " и размером " . $file['size'] . " байт");

}

?>

PHP автоматически сохраняет все загруженные файлы во временную папку на сервере. Но хранить там файлы нельзя, потому что эта директория периодически очищается, и ссылку на такой файл нельзя дать на сайте. Решение здесь только одно — переместить загруженный файл в другую папку. Перемещение файла всегда выполняют сразу после загрузки.

Для начала нужно убедиться, что в рабочей директории проекта существует папка для хранения загруженных файлов. Пусть она называется uploads.

**Перемещение загруженного файла**

Для перемещения файла нужно знать, где он находится сейчас, и адрес папки, в которую он будет переноситься.

С текущим адресом всё крайне просто — он уже находится в массиве $\_FILES. Новый адрес файла, в свою очередь, состоит из пути к папке и имени файла. Так как папка uploads находится там же, где и текущий сценарий, получить путь к ней можно так: dirname(\_\_FILE\_\_)`.

Код для перемещения файла в новую папку:

<?php

$current\_path = $\_FILES['avatar']['tmp\_name'];

$filename = $\_FILES['avatar']['name'];

$new\_path = dirname(\*\*\_\_FILE\_\_\*\*) . '/' . $filename;

move\_uploaded\_file($current\_path, $new\_path);

?>

Функция move\_uploaded\_file()выполняет два действия:

Проверяет, что файл действительно загружен через форму.

Перемещает загруженный файл по новому адресу.

**Валидация формы**

Валидация формы — это проверка содержимого её полей. Задача такой проверки — убедиться, что необходимые поля заполнены, а значения в них соответствуют ожидаемому формату.

Так, например, при регистрации пользователя на сайте, он должен заполнить поля с адресом электронной почты и придумать себе пароль. Оба поля обязательны к заполнению, но значение из поля email также должно быть корректным email-адресом.

Помимо текстовых значений формы, можно проверять формат и размер загружаемых файлов.

**Общий подход к валидации**

При выполнения валидации любой формы порядок действий будет всегда одним:

Сформировать массив с именами полей, обязательных к заполнению.

Сформировать массив с правилами для валидации формата полей.

Получить значения всех полей.

Создание функции PHP

Собственную PHP-функцию создать очень просто. Предположим, вы хотите создать функцию PHP, которая просто выводит в браузере короткое сообщение, когда вы ее вызываете. В следующем примере мы создаем функцию writeMessage(), а затем вызывает ее сразу после создания.

Обратите внимание, что при создании функции ее имя должно начинаться с ключевого слова function, и весь код PHP должен быть помещен внутри скобок { }, как в приведенном ниже примере:

<html>

  <head>

     <title>Writing PHP Function</title>

  </head>

  <body>

     <?php

        /\* Определение функции  PHP  \*/

        function writeMessage() {

           echo "You are really a nice person, Have a nice time!";

        }

        /\*Вызов функции PHP \*/

        writeMessage();

     ?>

  </body>

</html>

Этот код выводит следующий результат — You are really a nice person, Have a nice time!

Функции PHP с параметрами

PHP дает вам возможность передавать собственные параметры внутри функции. Вы можете передать столько параметров, сколько вам нужно. Эти параметры работают как переменные внутри функции. В следующем примере мы берем два целочисленных параметра и слагаем их, а затем выводим.

<html>

  <head>

     <title>Writing PHP Function with Parameters</title>

  </head>

  <body>

     <?php

        function addFunction($num1, $num2) {

           $sum = $num1 + $num2;

           echo "Sum of the two numbers is : $sum";

        }

        addFunction(10, 20);

     ?>

  </body>

</html>

Этот код выводит следующий результат — Sum of the two numbers is : 30

Передача аргументов по ссылке

В функции можно передавать аргументы по ссылке. Это означает, что ссылка на переменную управляется функцией, а не экземпляром значения переменной.

Любые изменения, внесенные в аргумент в этих случаях, изменяют значения исходной переменной. Вы можете передать аргумент по ссылке, добавив к имени переменной амперсанд (&) либо в вызове функции, либо в определении функции. В следующем примере продемонстрированы оба случая.

<html>

  <head>

     <title>Passing Argument by Reference</title>

  </head>

  <body>

     <?php

        function addFive($num) {

           $num += 5;

        }

        function addSix(&$num) {

           $num += 6;

        }

        $orignum = 10;

        addFive( $orignum );

        echo "Original Value is $orignum<br />";

        addSix( $orignum );

        echo "Original Value is $orignum<br />";

     ?>

  </body>

</html>

Этот код отображает следующий результат —

Original Value is 10

Original Value is 16

Функции PHP, возвращающие значение

Функция может возвращать значение с помощью оператора return в сочетании со значением или объектом. Return останавливает выполнение функции и отправляет значение обратно вызывающему коду. Более одного значения вы можете вернуть из функции с использованием массива return (1,2,3,4).

В следующем примере мы принимаем два целочисленных параметра и слагаем их, а затем возвращаем сумму в вызывающую программу. Обратите внимание, что ключевое слово return используется для возврата значения из функции.

<html>

  <head>

     <title>Writing PHP Function which returns value</title>

  </head>

  <body>

     <?php

        function addFunction($num1, $num2) {

           $sum = $num1 + $num2;

           return $sum;

        }

        $return\_value = addFunction(10, 20);

        echo "Returned value from the function : $return\_value";

     ?>

  </body>

</html>

Этот код выводит следующий результат: Returned value from the function : 30

Установка для параметров функций значений по умолчанию

Вы можете установить для параметра значение по умолчанию, если вызывающий функцию объект не передает его. Следующая функция выводит NULL в случае, если в эту функцию не было передано ни одного значения.  
<html>

  <head>

     <title>Writing PHP Function which returns value</title>

  </head>

  <body>

     <?php

        function printMe($param = NULL) {

           print $param;

        }

        printMe("This is test");

        printMe();

     ?>

  </body>

</html>

Это код выводит следующий результат — This is test

Динамические вызовы функций

Имена функций можно назначать, как строки в переменных, а затем обрабатывать эти переменные точно так же, как и имя функции. В следующем пример продемонстрировано такое поведение.

<html>

  <head>

     <title>Dynamic Function Calls</title>

  </head>

  <body>

     <?php

        function sayHello() {

           echo "Hello<br />";

        }

        $function\_holder = "sayHello";

        $function\_holder();

     ?>

  </body>

</html>

Этот код выводит следующий результат: Hello

**Функции PHP для работы с БД MySQL**

PHP имеет целый ряд встроенных функций для работы с СУБД MySQL.

**Соединение с сервером**

mysql\_connect (имя\_сервера [:номер\_порта], имя\_пользователя, пароль)

mysql\_pconnect (те же аргументы)

Первым аргументом может быть имя или адрес. Номер порта указывается, если он отличается от стандартного (3306). Функция возвращает идентификатор соединения.

Функция *mysql\_pconnect()* устанавливает устойчивое соединение, которое не разрывается после завершения работы сценария, а *mysql\_connect()* – неустойчивое. Если другой PHP-сценарий сделает вызов функции *mysql\_pconnect()* с аналогичными аргументами, то устойчивое соединение будет использовано повторно.

**Выбор базы данных**

mysql\_select\_db(имя\_бд)

**Выполнение указанного запроса**

mysql\_query(имя\_переменной с текстом запроса, имя\_переменной с идентификатором соединения)

Функция принимает в качестве аргумента текст SQL-запроса плюс идентификатор соединения, полученный от функции *mysql\_connect(),* и возвращает идентификатор результирующего набора записей. В случае ошибки возвращается 0.

**Извлечение информации**

о столбцах (полях):

mysql\_fetch\_field(имя\_переменной с идентификатором запроса)

Функция возвращает объект, содержащий описание полей указанного набора записей. В свойстве *name* этого объекта содержится вся информация о столбце, включая его тип и размерность.

Получить значения из записей (строк) можно разными способами:

mysql\_fetch\_row (имя\_переменной с идентификатором запроса)

Функция возвращает массив значений столбцов, индексация которых начинается с 0.

Для обхода всех записей в наборе данных (массиве) можно использовать оператор цикла *foreach*, который обеспечивает последовательный переход от записи к записи и присвоение значения каждого следующего элемента массива новой переменной:

foreach (имя\_переменной с результатом извлечения записи as перем\_для вывода на экран)

mysql\_fetch\_array(имя\_переменной с идентификатором запроса)

Функция преобразует результат запроса в обычный массив (по умолчанию и индексный, и ассоциативный)

**Ввод данных**

Для ввода данных необходимо выполнить запрос на вставку:

INSERT INTO назв\_таблицы (название\_поля\_1, назв\_поля\_2, ….., назв\_поля\_N) VALUES (‘значение\_1’, ‘знач\_1’, ……., знач\_N’).

Если вставка данных производится во все поля таблицы, то названия полей можно не перечислять, запрос будет выглядеть следующим образом:

INSERT INTO назв\_таблицы VALUES (‘значение\_1’, ‘знач\_1’, ……., знач\_N’).

Если вставка данных производится в поле типа AUTO\_INCREMENT, то вводится пустое значение, т.е. кавычки открываются и сразу закрываются,

**Основные причины поддержки нескольких соединений. Установка нескольких соединений. Синтаксические конструкции для контроля ошибок. Решение задачи контроля ошибок посредством функции die.**

Соединение (JOIN) - одна из самых важных операций, выполняемых реляционными системами управления базами данных (РСУБД). РСУБД используют соединения для того, чтобы сопоставить строки одной таблицы строкам другой таблицы. Например, соединения можно использовать для сопоставления продаж - клиентам или книг - авторам. Без соединений, имелись бы раздельные списки продаж и клиентов или книг и авторов, но невозможно было бы определить, какие клиенты что купили, или какой из авторов был заказан.

Можно соединить две таблицы явно, перечислив обе таблицы в предложении FROM запроса. Также можно соединить две таблицы, используя для этого всё разнообразие подзапросов. Наконец, SQL Server во время оптимизации может добавить соединение в план запроса, преследуя свои цели.

Это первая из серии статей, которые я планирую посвятить соединениям. Эту статью я собираюсь посвятить азам соединений, описав назначение логических операторов соединениё, поддерживаемых SQL Server. Вот они:

Inner join

Outer join

Cross join

Cross apply

Semi-join

Anti-semi-join

Для иллюстрации каждого соединения я буду использовать простую схему и набор данных

**Внутренние соединения**

Внутренние соединения - самый распространённый тип соединений. Внутреннее соединение просто находит пары строк, которые соединяются и удовлетворяют предикату соединения. Например, показанный ниже запрос использует предикат соединения "S.Cust\_Id = C.Cust\_Id", позволяющий найти все продажи и сведения о клиенте с одинаковыми значениями Cust\_Id:

**Внешние соединения**

Предположим, что мы хотели бы увидеть список всех продаж; даже тех, которые не имеют соответствующих им записей о клиенте. Можно составить запрос с внешним соединением, которое покажет все строки в одной или обеих соединяемых таблицах, даже если не будет существовать соответствующих предикату соединения строку. Например:

Обратите внимание, что сервер возвращает вместо данных о клиенте значение NULL, поскольку для проданного товара 'Printer' нет соответствующей записи клиента. Обратите внимание на последнюю строку, у которой отсутствующие значения заполнены значением NULL.

Используя полное внешнее соединение, можно найти всех клиентов (независимо от того, покупали ли они что-нибудь), и все продажи (независимо от того, сопоставлен ли им имеющийся клиент):

**Перекрестные соединения**

Перекрестное соединение выполняет полное Декартово произведение двух таблиц. То есть это соответствие каждой строки одной таблицы - каждой строке другой таблицы. Для перекрестного соединения нельзя определить предикат соединения, используя для этого предложение ON, хотя для достижения практически того же результата, что и с внутренним соединением, можно использовать предложение WHERE.

Перекрестные соединения используются довольно редко. Никогда не стоит пересекать две большие таблицы, поскольку это задействует очень дорогие операции и получится очень большой результирующий набор.

**CROSS APPLY**

В SQL Server 2005 мы добавили оператор CROSS APPLY, с помощью которого можно соединять таблицу с возвращающей табличное значение функцией (table valued function - TVF), причём TVF будет иметь параметр, который будет изменяться для каждой строки. Например, представленный ниже запрос возвратит тот же результат, что и показанное ранее внутреннее соединение, но с использованием TVF и CROSS APPLY:

**create** **function** dbo.fn\_Sales(@Cust\_Id int)

**returns** @Sales **table** (Item varchar(10))

**as**

**begin**

**insert** @Sales **select** Item **from** Sales **where** Cust\_Id = @Cust\_Id

**return**

**end**

**select** \*

**from** Customers **cross** apply dbo.fn\_Sales(Cust\_Id)

Cust\_Id Cust\_Name Item

----------- ---------- ----------

2 John Doe Camera

3 Jane Doe Computer

3 Jane Doe Monitor

Также можно использовать внешнее обращение - OUTER APPLY, позволяющее нам найти всех клиентов независимо от того, купили ли они что-нибудь или нет. Это будет похоже на внешнее соединение.

**select** \*

**from** Customers **outer** apply dbo.fn\_Sales(Cust\_Id)

Cust\_Id Cust\_Name Item

----------- ---------- ----------

1 Craig **NULL**

2 John Doe Camera

3 Jane Doe Computer

3 Jane Doe Monitor

**Полусоединение и анти-полусоединение**

Полусоединение - semi-join возвращает строки только одной из соединяемых таблиц, без выполнения соединения полностью. Анти-полусоединение возвращает те строки таблицы, которые не годятся для соединения с другой таблицей; т.е. они в обычном внешнем соединении выдавали бы NULL.

В отличие от других операторов соединений, не существует явного синтаксиса для указания исполнения полусоединения, но SQL Server, в целом ряде случаев, использует в плане исполнения именно полусоединения.

Существуют левые и правые полусоединения. Левое полусоединение возвращает строки левой (первой) таблицы, которые соответствуют строкам из правой (второй) таблицы, в то время как правое полусоединение возвращает строки из правой таблицы, которые соответствуют строкам из левой таблицы.

Подобным образом может использоваться анти-полусоединение для обработки подзапроса с NOT EXISTS.

Почему мы обрабатываем ошибки в нашем коде? На это есть много причин. Например, на формах в приложении мы проверяем введенные данные и информируем пользователей о допущенных при вводе ошибках. Ошибки пользователя – это предвиденные ошибки. Но нам также нужно обрабатывать непредвиденные ошибки. То есть, ошибки могут возникнуть из-за того, что мы что-то упустили при написании кода. Простой подход – это прервать выполнение или хотя бы вернуться на этап, в котором мы имеем полный контроль над происходящим. Недостаточно будет просто подчеркнуть, что совершенно непозволительно игнорировать непредвиденные ошибки. Это недостаток, который может вызвать губительные последствия: например, стать причиной того, что приложение будет предоставлять некорректную информацию пользователю или, что еще хуже, сохранять некорректные данные в базе. Также важно сообщать о возникновении ошибки с той целью, чтобы пользователь не думал о том, что операция прошла успешно, в то время как ваш код на самом деле ничего не выполнил.  
  
Мы часто хотим, чтобы в базе данных изменения были [атомарными](https://ru.wikipedia.org/wiki/%D0%90%D1%82%D0%BE%D0%BC%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F). Например, задача по переводу денег с одного счета на другой. С этой целью мы должны изменить две записи в таблице CashHoldings и добавить две записи в таблицу Transactions. Абсолютно недопустимо, чтобы ошибки или сбой привели к тому, что деньги будут переведены на счет получателя, а со счета отправителя они не будут списаны. По этой причине обработка ошибок также касается и обработки транзакций. В приведенном примере нам нужно обернуть операцию в BEGIN TRANSACTION и COMMIT TRANSACTION, но не только это: в случае ошибки мы должны убедиться, что транзакция откачена.

2. Основные команды

Мы начнем с обзора наиболее важных команд, которые необходимы для обработки ошибок. Во второй части я опишу все команды, относящиеся к обработке ошибок и транзакций.

2.1 TRY-CATCH

Основным механизмом обработки ошибок является конструкция TRY-CATCH, очень напоминающая подобные конструкции в других языках. Структура такова:

BEGIN TRY

<обычный код>

END TRY

BEGIN CATCH

<обработка ошибок>

END CATCH

Если какая-либо ошибка появится в <обычный код>, выполнение будет переведено в блок CATCH, и будет выполнен код обработки ошибок.  
  
Как правило, в CATCH откатывают любую открытую транзакцию и повторно вызывают ошибку. Таким образом, вызывающая клиентская программа понимает, что что-то пошло не так. Повторный вызов ошибки мы обсудим позже в этой статье.  
  
Вот очень быстрый пример:

BEGIN TRY

DECLARE @x int

SELECT @x = 1/0

PRINT 'Not reached'

END TRY

BEGIN CATCH

PRINT 'This is the error: ' + error\_message()

END CATCH

Результат выполнения: This is the error: Divide by zero error encountered.  
  
Мы вернемся к функции error\_message() позднее. Стоит отметить, что использование PRINT в обработчике CATCH приводится только в рамках экспериментов и не следует делать так в коде реального приложения.  
  
Если <обычный код> вызывает хранимую процедуру или запускает триггеры, то любая ошибка, которая в них возникнет, передаст выполнение в блок CATCH. Если более точно, то, когда возникает ошибка, SQL Server раскручивает стек до тех пор, пока не найдёт обработчик CATCH. И если такого обработчика нет, SQL Server отправляет сообщение об ошибке напрямую клиенту.  
  
Есть одно очень важное ограничение у конструкции TRY-CATCH, которое нужно знать: она не ловит ошибки компиляции, которые возникают в той же области видимости. Рассмотрим пример:

CREATE PROCEDURE inner\_sp AS

BEGIN TRY

PRINT 'This prints'

SELECT \* FROM NoSuchTable

PRINT 'This does not print'

END TRY

BEGIN CATCH

PRINT 'And nor does this print'

END CATCH

go

EXEC inner\_sp

Выходные данные:

This prints

Msg 208, Level 16, State 1, Procedure inner\_sp, Line 4

Invalid object name 'NoSuchTable'

Как можно видеть, блок TRY присутствует, но при возникновении ошибки выполнение не передается блоку CATCH, как это ожидалось. Это применимо ко всем ошибкам компиляции, таким как пропуск колонок, некорректные псевдонимы и тому подобное, которые возникают во время выполнения. (Ошибки компиляции могут возникнуть в SQL Server во время выполнения из-за отложенного разрешения имен – особенность, благодаря которой SQL Server позволяет создать процедуру, которая обращается к несуществующим таблицам.)  
  
Эти ошибки не являются полностью неуловимыми; вы не можете поймать их в области, в которой они возникают, но вы можете поймать их во внешней области. Добавим такой код к предыдущему примеру:

CREATE PROCEDURE outer\_sp AS

BEGIN TRY

EXEC inner\_sp

END TRY

BEGIN CATCH

PRINT 'The error message is: ' + error\_message()

END CATCH

go

EXEC outer\_sp

Теперь мы получим на выходе это:

This prints

The error message is: Invalid object name 'NoSuchTable'.

На этот раз ошибка была перехвачена, потому что сработал внешний обработчик CATCH.

2.2 SET XACT\_ABORT ON

В начало ваших хранимых процедур следует всегда добавлять это выражение:

SET XACT\_ABORT, NOCOUNT ON

Оно активирует два параметра сессии, которые выключены по умолчанию в целях совместимости с предыдущими версиями, но опыт доказывает, что лучший подход – это иметь эти параметры всегда включенными. Поведение SQL Server по умолчанию в той ситуации, когда не используется TRY-CATCH, заключается в том, что некоторые ошибки прерывают выполнение и откатывают любые открытые транзакции, в то время как с другими ошибками выполнение последующих инструкций продолжается. Когда вы включаете XACT\_ABORT ON, почти все ошибки начинают вызывать одинаковый эффект: любая открытая транзакция откатывается, и выполнение кода прерывается. Есть несколько исключений, среди которых наиболее заметным является выражение RAISERROR.  
  
Параметр XACT\_ABORT необходим для более надежной обработки ошибок и транзакций. В частности, при настройках по умолчанию есть несколько ситуаций, когда выполнение может быть прервано без какого-либо отката транзакции, даже если у вас есть TRY-CATCH. Мы видели такой пример в предыдущем разделе, где мы выяснили, что TRY-CATCH не перехватывает ошибки компиляции, возникшие в той же области. Открытая транзакция, которая не была откачена из-за ошибки, может вызвать серьезные проблемы, если приложение работает дальше без завершения транзакции или ее отката.  
  
Для надежной обработки ошибок в SQL Server вам необходимы как TRY-CATCH, так и SET XACT\_ABORT ON. Среди них инструкция SET XACT\_ABORT ON наиболее важна. Если для кода на промышленной среде только на нее полагаться не стоит, то для быстрых и простых решений она вполне подходит.  
  
Параметр NOCOUNT не имеет к обработке ошибок никакого отношения, но включение его в код является хорошей практикой. NOCOUNT подавляет сообщения вида (1 row(s) affected), которые вы можете видеть в панели Message в SQL Server Management Studio. В то время как эти сообщения могут быть полезны при работе c SSMS, они могут негативно повлиять на производительность в приложении, так как увеличивают сетевой трафик. Сообщение о количестве строк также может привести к ошибке в плохо написанных клиентских приложениях, которые могут подумать, что это данные, которые вернул запрос.  
  
Выше я использовал синтаксис, который немного необычен. Большинство людей написали бы два отдельных выражения:

SET NOCOUNT ON

SET XACT\_ABORT ON

Между ними нет никакого отличия. Я предпочитаю версию с SET и запятой, т.к. это снижает уровень шума в коде. Поскольку эти выражения должны появляться во всех ваших хранимых процедурах, они должны занимать как можно меньше места.

3. Основной пример обработки ошибок

После того, как мы посмотрели на TRY-CATCH и SET XACT\_ABORT ON, давайте соединим их вместе в примере, который мы можем использовать во всех наших хранимых процедурах. Для начала я покажу пример, в котором ошибка генерируется в простой форме, а в следующем разделе я рассмотрю решения получше.

Первая строка в процедуре включает XACT\_ABORT и NOCOUNT в одном выражении, как я показывал выше. Эта строка – единственная перед BEGIN TRY. Все остальное в процедуре должно располагаться после BEGIN TRY: объявление переменных, создание временных таблиц, табличных переменных, всё. Даже если у вас есть другие SET-команды в процедуре (хотя причины для этого встречаются редко), они должны идти после BEGIN TRY.  
  
Причина, по которой я предпочитаю указывать SET XACT\_ABORT и NOCOUNT перед BEGIN TRY, заключается в том, что я рассматриваю это как одну строку шума: она всегда должна быть там, но я не хочу, чтобы это мешало взгляду. Конечно же, это дело вкуса, и если вы предпочитаете ставить SET-команды после BEGIN TRY, ничего страшного. Важно то, что вам не следует ставить что-либо другое перед BEGIN TRY.  
  
Часть между BEGIN TRY и END TRY является основной составляющей процедуры. Поскольку я хотел использовать транзакцию, определенную пользователем, я ввел довольно надуманное бизнес-правило, в котором говорится, что если вы вставляете пару, то обратная пара также должна быть вставлена. Два выражения INSERT находятся внутри BEGIN и COMMIT TRANSACTION. Во многих случаях у вас будет много строк кода между BEGIN TRY и BEGIN TRANSACTION. Иногда у вас также будет код между COMMIT TRANSACTION и END TRY, хотя обычно это только финальный SELECT, возвращающий данные или присваивающий значения выходным параметрам. Если ваша процедура не выполняет каких-либо изменений или имеет только одно выражение INSERT/UPDATE/DELETE/MERGE, то обычно вам вообще не нужно явно указывать транзакцию.  
  
В то время как блок TRY будет выглядеть по-разному от процедуры к процедуре, блок CATCH должен быть более или менее результатом копирования и вставки. То есть вы делаете что-то короткое и простое и затем используете повсюду, не особо задумываясь. Обработчик CATCH, приведенный выше, выполняет три действия:

Откатывает любые открытые транзакции.

Повторно вызывает ошибку.

Убеждается, что возвращаемое процедурой значение отлично от нуля.

Эти три действия должны всегда быть там. Мы можете возразить, что строка

IF @@trancount > 0 ROLLBACK TRANSACTION

не нужна, если нет явной транзакции в процедуре, но это абсолютно неверно. Возможно, вы вызываете хранимую процедуру, которая открывает транзакцию, но которая не может ее откатить из-за ограничений TRY-CATCH. Возможно, вы или кто-то другой добавите явную транзакцию через два года. Вспомните ли вы тогда о том, что нужно добавить строку с откатом? Не рассчитывайте на это. Я также слышу читателей, которые возражают, что если тот, кто вызывает процедуру, открыл транзакцию, мы не должны ее откатывать… Нет, мы должны, и если вы хотите знать почему, вам нужно прочитать вторую и третью части. Откат транзакции в обработчике CATCH – это категорический императив, у которого нет исключений.  
  
Код повторной генерации ошибки включает такую строку:

DECLARE @msg nvarchar(2048) = error\_message()

Встроенная функция error\_message() возвращает текст возникшей ошибки. В следующей строке ошибка повторно вызывается с помощью выражения RAISERROR. Это не самый простой способ вызова ошибки, но он работает. Другие способы мы рассмотрим в следующей главе.  
  
Замечание: синтаксис для присвоения начального значения переменной в DECLARE был внедрен в SQL Server 2008. Если у вас SQL Server 2005, вам нужно разбить строку на DECLARE и выражение SELECT.  
  
Финальное выражение RETURN – это страховка. RAISERROR никогда не прерывает выполнение, поэтому выполнение следующего выражения будет продолжено. Пока все процедуры используют TRY-CATCH, а также весь клиентский код обрабатывает исключения, нет повода для беспокойства. Но ваша процедура может быть вызвана из старого кода, написанного до SQL Server 2005 и до внедрения TRY-CATCH. В те времена лучшее, что мы могли делать, это смотреть на возвращаемые значения. То, что вы возвращаете с помощью RETURN, не имеет особого значения, если это не нулевое значение (ноль обычно обозначает успешное завершение работы).  
  
Последнее выражение в процедуре – это END CATCH. Никогда не следует помещать какой-либо код после END CATCH. Кто-нибудь, читающий процедуру, может не увидеть этот кусок кода.  
  
После прочтения теории давайте попробуем тестовый пример:

EXEC insert\_data 9, NULL

Результат выполнения:

Msg 50000, Level 16, State 1, Procedure insert\_data, Line 12

Cannot insert the value NULL into column 'b', table 'tempdb.dbo.sometable'; column does not allow nulls. INSERT fails.

Давайте добавим внешнюю процедуру для того, чтобы увидеть, что происходит при повторном вызове ошибки:

CREATE PROCEDURE outer\_sp @a int, @b int AS

SET XACT\_ABORT, NOCOUNT ON

BEGIN TRY

EXEC insert\_data @a, @b

END TRY

BEGIN CATCH

IF @@trancount > 0 ROLLBACK TRANSACTION

DECLARE @msg nvarchar(2048) = error\_message()

RAISERROR (@msg, 16, 1)

RETURN 55555

END CATCH

go

EXEC outer\_sp 8, 8

Результат работы:

Msg 50000, Level 16, State 1, Procedure outer\_sp, Line 9

Violation of PRIMARY KEY constraint 'pk\_sometable'. Cannot insert duplicate key in object 'dbo.sometable'. The duplicate key value is (8, 8).

Мы получили корректное сообщение об ошибке, но если вы посмотрите на заголовки этого сообщения и на предыдущее поближе, то можете заметить проблему:

Msg 50000, Level 16, State 1, Procedure insert\_data, Line 12

Msg 50000, Level 16, State 1, Procedure outer\_sp, Line 9

Сообщение об ошибке выводит информацию о расположении конечного выражения RAISERROR. В первом случае некорректен только номер строки. Во втором случае некорректно также имя процедуры. Для простых процедур, таких как наш тестовый пример, это не является большой проблемой. Но если у вас есть несколько уровней вложенных сложных процедур, то наличие сообщения об ошибке с отсутствием указания на место её возникновения сделает поиск и устранение ошибки намного более сложным делом. По этой причине желательно генерировать ошибку таким образом, чтобы можно было определить нахождение ошибочного фрагмента кода быстро, и это то, что мы рассмотрим в следующей главе.

4. Три способа генерации ошибки

4.1 Использование error\_handler\_sp

Мы рассмотрели функцию error\_message(), которая возвращает текст сообщения об ошибке. Сообщение об ошибке состоит из нескольких компонентов, и существует своя функция error\_xxx() для каждого из них. Мы можем использовать их для повторной генерации полного сообщения, которое содержит оригинальную информацию, хотя и в другом формате. Если делать это в каждом обработчике CATCH, это будет большой недостаток — дублирование кода. Вам не обязательно находиться в блоке CATCH для вызова error\_message() и других подобных функций, и они вернут ту же самую информацию, если будут вызваны из хранимой процедуры, которую выполнит блок CATCH.

Первое из того, что делает error\_handler\_sp – это сохраняет значение всех error\_xxx() функций в локальные переменные. Я вернусь к выражению IF через секунду.

Цель этого SELECT заключается в форматировании сообщения об ошибке, которое передается в RAISERROR. Оно включает в себя всю информацию из оригинального сообщения об ошибке, которое мы не можем вставить напрямую в RAISERROR. Мы должны обработать имя процедуры, которое может быть NULL для ошибок в обычных скриптах или в динамическом SQL. Поэтому используется функция COALESCE. (Если вы не понимаете форму выражения RAISERROR, я рассказываю о нем более детально во второй части.)  
  
Отформатированное сообщение об ошибке начинается с трех звездочек. Этим достигаются две цели: 1) Мы можем сразу видеть, что это сообщение вызвано из обработчика CATCH. 2) Это дает возможность для error\_handler\_sp отфильтровать ошибки, которые уже были сгенерированы один или более раз, с помощью условия NOT LIKE ‘\*\*\*%’ для того, чтобы избежать изменения сообщения во второй раз.  
  
Вот как обработчик CATCH должен выглядеть, когда вы используете error\_handler\_sp:

BEGIN CATCH

IF @@trancount > 0 ROLLBACK TRANSACTION

EXEC error\_handler\_sp

RETURN 55555

END CATCH

Заголовки сообщений говорят о том, что ошибка возникла в процедуре error\_handler\_sp, но текст сообщений об ошибках дает нам настоящее местонахождение ошибки – как название процедуры, так и номер строки.  
  
Я покажу еще два метода вызова ошибок. Однако error\_handler\_sp является моей главной рекомендацией для читателей, которые читают эту часть. Это — простой вариант, который работает на всех версиях SQL Server начиная с 2005. Существует только один недостаток: в некоторых случаях SQL Server генерирует два сообщения об ошибках, но функции error\_xxx() возвращают только одну из них, и поэтому одно из сообщений теряется. Это может быть неудобно при работе с административными командами наподобие BACKUP\RESTORE, но проблема редко возникает в коде, предназначенном чисто для приложений.

4.2. Использование ;THROW

В SQL Server 2012 Microsoft представил выражение ;THROW для более легкой обработки ошибок. К сожалению, Microsoft сделал серьезную ошибку при проектировании этой команды и создал опасную ловушку.  
  
С выражением ;THROW вам не нужно никаких хранимых процедур. Ваш обработчик CATCH становится таким же простым, как этот:

BEGIN CATCH

IF @@trancount > 0 ROLLBACK TRANSACTION

;THROW

RETURN 55555

END CATCH

Достоинство ;THROW в том, что сообщение об ошибке генерируется точно таким же, как и оригинальное сообщение. Если изначально было два сообщения об ошибках, оба сообщения воспроизводятся, что делает это выражение еще привлекательнее. Как и со всеми другими сообщениями об ошибках, ошибки, сгенерированные ;THROW, могут быть перехвачены внешним обработчиком CATCH и воспроизведены. Если обработчика CATCH нет, выполнение прерывается, поэтому оператор RETURN в данном случае оказывается не нужным. (Я все еще рекомендую оставлять его, на случай, если вы измените свое отношение к ;THROW позже).  
  
Если у вас SQL Server 2012 или более поздняя версия, измените определение insert\_data и outer\_sp и попробуйте выполнить тесты еще раз.

Имя процедуры и номер строки верны и нет никакого другого имени процедуры, которое может нас запутать. Также сохранены оригинальные номера ошибок.  
  
  
Нельзя отрицать того, что ;THROW имеет свои преимущества, но точка с запятой не единственная ловушка этой команды. Если вы хотите использовать ее, я призываю вас прочитать по крайней мере вторую часть этой серии, где я раскрываю больше деталей о команде ;THROW. До этого момента, используйте error\_handler\_sp.  
Третий способ обработки ошибок – это использование SqlEventLog, который я описываю очень детально в третьей части. Здесь я лишь сделаю короткий обзор.  
  
SqlEventLog предоставляет хранимую процедуру slog.catchhandler\_sp, которая работает так же, как и error\_handler\_sp: она использует функции error\_xxx() для сбора информации и выводит сообщение об ошибке, сохраняя всю информацию о ней. Вдобавок к этому, она логирует ошибку в таблицу splog.sqleventlog. В зависимости от типа приложения, которое у вас есть, эта таблица может быть очень ценным объектом.  
  
Для использования SqlEventLog, ваш обработчик CATCH должен быть таким:

BEGIN CATCH

IF @@trancount > 0 ROLLBACK TRANSACTION

EXEC slog.catchhandler\_sp @@procid

RETURN 55555

END CATCH

@@procid возвращает идентификатор объекта текущей хранимой процедуры. Это то, что SqlEventLog использует для логирования информации в таблицу.

**Общие особенности трехзвенной архитектуры. Способы программной реализации трехзвенной архитектуры**

В компьютерных технологиях **трёхуровневая архитектура**, синоним **трёхзвенная архитектура** (*англ*. three-tier или Multitier architecture) предполагает наличие следующих компонентов приложения: клиентское приложение (обычно говорят «тонкий клиент» или терминал), подключенное к серверу приложений, который в свою очередь подключен к серверу базы данных.

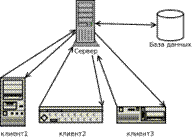
Обзор архитектуры

· *Клиент* — это интерфейсный (обычно графический) компонент, который представляет первый уровень, собственно приложение для конечного пользователя. Первый уровень не должен иметь прямых связей с базой данных (по требованиям безопасности), быть нагруженным основной бизнес-логикой (по требованиям масштабируемости) и хранить состояние приложения (по требованиям надежности). На первый уровень может быть вынесена и обычно выносится простейшая бизнес-логика: интерфейс авторизации, алгоритмы шифрования, проверка вводимых значений на допустимость и соответствие формату, несложные операции (сортировка, группировка, подсчет значений) с данными, уже загруженными на терминал.

|  |  |
| --- | --- |
|  |  |

· *Сервер приложений* располагается на втором уровне. На втором уровне сосредоточена бо́льшая часть бизнес-логики. Вне его остаются фрагменты, экспортируемые на терминалы (см.выше), а также погруженные в третий уровень хранимые процедуры и триггеры.

· *Сервер базы данных* обеспечивает хранение данных и выносится на третий уровень. Обычно это стандартная реляционная или объектно-ориентированная СУБД. Если третий уровень представляет собой базу данных вместе с хранимыми процедурами, триггерами и схемой, описывающей приложение в терминах реляционной модели, то второй уровень строится как программный интерфейс, связывающий клиентские компоненты с прикладной логикой базы данных.



В простейшей конфигурации физически сервер приложений может быть совмещён с сервером базы данных на одном компьютере, к которому по сети подключается один или несколько терминалов.

В «правильной» (с точки зрения безопасности, надёжности, масштабирования) конфигурации сервер базы данных находится на выделенном компьютере (или кластере), к которому по сети подключены один или несколько серверов приложений, к которым, в свою очередь, по сети подключаются терминалы.

**Транзакция**– это непрерывное управление данными, которое переводит БД из одного целостного состояния в другое.

В принципе, архитектура приложений "клиент-сервер" не ограничивается тремя уровнями и может быть многоуровневой. Многоуровневая архитектура предполагает дальнейшую декомпозицию приложения на функционально различные компоненты и распределение этих компонентов по узлам вычислительной сети. Специалисты считают, что использовать трехуровневую архитектуру целесообразно в следующих ситуациях, когда:

число параллельных подключений к БД превышает 50;

приложение вызывает функции, которые одновременно необходимы разным модулям приложения (эти модули реализуют определённую логику системы и представляют собой хранимые процедуры);

применяется кэширование данных;

**Кэширование**– приём, при котором информация, часто используемая и хранящаяся во внешней памяти, временно записывается в быструю оперативную память. Когда эта информация необходима, она считывается из оперативной памяти.

приложение должно функционировать в глобальной сети с территориально распределенными узлами;

требуется проверка полномочий пользователей, обращающихся к БД. При этом трехуровневая архитектура снижает издержки на управление защитой и её обслуживание.

В трехуровневой архитектуре клиентское программное обеспечение не обращается непосредственно к СУБД и освобождается от большей части обрабатывающих программ, которые переносятся на сервер приложения. В результате получается *облегченный* или *тонкий клиент*, который не требует больших ресурсов памяти и может загружаться с более мощного сетевого компьютера.

Сервер приложений реализовывался с помощью компонентов, соответствующих выбранной технологии. Каждый из этих компонентов представляет собой окно-контейнер, в который помещаются компоненты для связи с удаленной БД, компоненты источники данных, а также объекты поля для соответствующих наборов данных.

Сервер приложений. Технологии удаленного доступа. Создание сервера приложений. Управление данными. Клиентское приложение. Виды связи. Управление связью. Раннее и позднее связывание с интерфейсом сервера.

**Сервер приложений** (англ. *application server*) — это программная платформа, предназначенная для эффективного исполнения процедур (программ, скриптов), на которых построены приложения. Сервер приложений действует как набор компонентов, доступных разработчику программного обеспечения через API (интерфейс прикладного программирования), определённый самой платформой.

Для веб-приложений основная задача компонентов сервера — обеспечивать создание динамических страниц. Однако современные серверы приложений включают в себя и поддержку кластеризации, повышенную отказоустойчивость, балансировку нагрузки, позволяя таким образом разработчикам сфокусироваться только на реализации бизнес-логики.

В случае Java-сервера приложений сервер приложений ведёт себя как расширенная виртуальная машина для запуска приложений, прозрачно управляя соединениями с базой данных, с одной стороны, и соединениями с веб-клиентом, с другой.

Удалённый доступ — это технология, позволяющая пользователю подключиться к компьютеру на расстоянии с помощью другого устройства и управлять им так, как если бы он сам сидел за этим компьютером. Это может быть и подключение к рабочей машине из дома, и помощь другу с установкой программ — вариантов много. Главное условие — наличие на обоих устройствах выхода в интернет и специальной программы.

Для организации удалённого доступа существует множество программ, как сложных, так и очень простых, некоторые из которых мы постараемся осветить.

Устройство в сети удалённого доступа может играть две роли:

хост — компьютер, к которому предоставляется доступ (например, рабочий компьютер, на котором нужно поработать из дома);

клиент — машина, осуществляющая доступ к другим устройствам.

Один и тот же компьютер в разных ситуациях может использоваться и как хост, и как клиент — но не одновременно.

Каждому устройству присваивается свой идентификатор — метка, с помощью которой устройства «находят» друг друга в сети. В большинстве случаев подключение происходит так: клиенту выдаётся идентификатор хоста, тот находит его в сети и подключается к нему, предоставляя пользователю доступ. В целях защиты также может потребоваться уникальный одноразовый пароль, который виден только хосту: это делается, чтобы к компьютеру не мог подключиться недоверенный человек.

Управление данными включает в себя процессы переработки данных, начиная от сбора данных и заканчивая их архивацией и доведением до пользователей. При этом рассматриваются как технологические, так и организационные вопросы сбора, обработки данных. Администрирование БД – это компонент управления данными, связанный с СУБД.

Управление данными можно рассматривать на уровне источника данных, центра данных, проекта (программы). Каждый уровень может включать предыдущие уровни управления данными. Например, управление данными на уровне центра обязательно включает сбор данных от источников данных. Крупная научная программа может включать несколько экспериментов, каждый из которых может иметь свой план управления данными.

План управления данными – это организационный документ, в котором определены все этапы переработки данных, а также средства их реализации.

Целями создания плана управления данными является улучшение сбора, доступа и использования информации; развитие БД; стандартизация процедур сбора и обмена данными.

Создание плана управления данными должно учитывать долгопериодные решения по

развитию и стандартизации общих технологий сбора и обмена данными, позволяющих уменьшить временной лаг между сбором и доступом к данным;

увеличению кооперации при сборе, архивации, обработке и картированию данных;

созданию распределенных БД;

объединению новых и исторических данных для получения соответствующих временных рядов;

совместимости БД за счет использования общих протоколов форматирования и контроля качества для отдельных дисциплин;

доступу к архивным данным.

Методология управления данными должна быть основана на применении наиболее эффективных средств:

создания многоуровневых каталогов данных;

использования каталогов для поиска и оценки дубликатов;

поиска и обмена данными;

конвертирования данных в общие форматы;

контроля данными на различных этапах переработки данных;

создания новых методов обработки данных;

доступ к данным на компактных дисках, Интернет и др.

План управления данными способствует лучшему пониманию всеми участниками проекта, объединения научных интересов, общественных потребностей и правовых вопросов. Управление данными начинается с проектирования измерительной программы экспедиции или проекта, создания БД и заканчивается доступом пользователей к качественно проконтролированным и хорошо задокументированным БД. План управления данными должен быть ключевым элементом всех крупных проектов и программ. План управления данными поможет максимизировать возврат инвестиций, сделанных в проект с помощью финансирования для целей всестороннего использования получаемых данных, т.е. план управления данными есть механизм распространения и использования результатов проекта, специальная активность, выполняемая в рамках национальной и международной или корпоративной политики, основанной на лучшей практике обработки данных.

Этот план должен описывать работу, технологические требования и соответствующие результаты в проектировании измерительной активности, отчетности по сбору данных, документировании, контроле качества и создании БД, доступа к данным.

Одной из главных задач любого проекта, а особенно центра данных, является создание баз метаданных. Общие подходы в управлении данными позволяют получить пользу как специалистам, работающим в этих проектах, так и обществу в целом (более быстрое использование данных); сделать эффективнее использование большинства источников данных; хорошо задокументировать и проконтролировать данные, предназначенные для общего использования по окончании проекта.

Адекватное управление данными определяется возможностями национальных организаций политическими аспектами, техническими проблемами, условиями финансирования проектов, хорошей координацией всех участников проекта, наличием соответствующего квалифицированного штата.

План управления данными проекта отражает руководящие основы, необходимые для подготовки решений по переработке данных. Критериями управления данными в этом случае являются:

максимизация сбора данных с широким использованием согласованных стандартов;

хорошее документирование данных;

максимизация использования данных путем организации обмена данными;

последующее длительное использование данных национальным и международным сообществом путем различных публикаций, в т.ч. современными средствами в Интернет и на компактных дисках.

План управления данными формируется на этапе подготовки предложений по проекту в виде раздела «Управление данными» или самостоятельного документа. План управления данными должен отражать проектные решения по технологиям сбора данных, подходы к организации баз данных, используемые стандарты и другие, которые могут быть эффективно использованы в проекте. Сейчас для большинства проектов управление данными есть часть большой работы, для которой создаются специальные группы по отдельным дисциплинам или в рамках корпорации.

Документ «План управления данными» включает:

Введение.

Краткий обзор, состояния вопроса.

Описание имеющихся массивов данных.

Основная концепция управления данными.

Классификация, поиск и инвентаризация, сбор и обмен данными.

Методы обработки данных.

Программное обеспечение для обработки данных.

Сотрудничество (международное, межведомственное, корпоративное).

Использование локальных и глобальных сетей для управления данными.

Основные шаги реализации Плана управления данными (идентификация пользователей, участвующих организаций в исследовании региона и требований пользователей к информационному обеспечению; описание моделей анализа и прогноза, развитие новых моделей; описание методов использования моделей и других программных средств).

Кроме того, план управления данными представляет:

Потоки данных и информации,

Правила документирования данных,

Технологии оцифровки данных (занесения на технические носители),

Конвертирование данных для приведения в один формат хранения,

Требования по поиску, выборке, обработке и представлению данных на экране,

Возможности дальнейшего использования данных,

Методы архивации данных,

Организационные вопросы – кто, что, когда, в какие сроки создает и представляет пользователям ту или иную информацию; проведение семинаров, обучающих курсов и др.

**В разделе «Основная концепция управления данными»**рассматриваются принципы управления, политика управления данными, объект и функции управления, документирование данных, создание БД, использование информационных стандартов, контроль качества данных.

При разработке программного обеспечения используются общие решения: стандарты на методы регистрации, форматы данных, модельные данные, метаданные, интерфейс, термины и определения, языки описания данных и манипуляции.

Рекомендуется использовать современные инструментальные средства на всех стадиях обработки данных (сбора, накопления, хранения, вычисления, анализа, интерпретации и распространения данных). Важным моментом плана управления данными должны быть организационные правила (статус информационных технологий, включающий правила взаимодействия с информационными системами, обмена данными и т.п.).

**Основными шагами реализации плана управления данными являются:**

создание баз метаданных на Web;

организация поиска, обмена, контроля, хранения и распространения данных;

создание базы исходных данных;

поддержка прикладных задач;

разработка интегрированной базы данных.

К плану управления данными прилагаются списки комитетов, рабочих групп, участвующих организаций, описание выполняемых проектов; примеры описания форматов хранения и обмена данными, применяемых кодификаторов.

**Администрирование БД**

Относительно крупная БД (объемом более 1 Гбайта, с количеством логических единиц сбора и обработки данных в десятки тысяч) требует эффективного управления. Для этого существуют специальные лица, участвующие в эксплуатации БД **–** администраторы БД.

Типы администраторов и их обязанности могут отличаться в зависимости от конфигурации СУБД и от конкретной организации. В крупных системах обязанности администратора БД могут распределяться среди нескольких специалистов. В то же время в небольших системах один человек может выполнять функции нескольких типов одновременно.

Можно выделить основные типы администраторов, характерные для всех систем: администраторы баз данных, администраторы по защите данных; компьютерные администраторы; сетевые администраторы; администраторы Интернет (Web–мастера); администраторы телефонной связи; администраторы голосовой почты; администраторы почтовых систем; администраторы мэйнфреймов; администраторы приложений.

Администратор БД отвечает за целостность информационных ресурсов компании. На нем лежит ответственность по созданию, обновлению и сохранности связанных между собой резервных копий файлов, исходя из задач предприятия. Этот человек должен в мельчайших подробностях знать существующие механизмы восстановления программного обеспечения БД. Администратор БД должен уметь определять узкие места системы, ограничивающие ее производительность, настраивать SQL и программное обеспечение СУБД и обладать знаниями, необходимыми для решения вопросов оптимизации быстродействия БД.

Основные обязанности администратора БД являются стандартом для большинства систем, в то время как дополнительные могут варьироваться в зависимости от конкретной организации. Вот некоторые типы администраторов БД: оперативные, тактические, стратегические, прикладные, системные, наемные, администраторы – руководители. Обязанности администратора БД:

создание баз данных, табличных пространств, таблиц, представлений и индексов согласно спецификации разработчика приложений;

периодическая проверка производительности системы и произведение изменений для поддержки необходимого уровня производительности;

поддержка целостности данных БД;

планирование и выполнение качественного резервного копирования и стратегии восстановления;

установка нового программного обеспечения (очень важно протестировать все программы перед введением их в рабочую среду);

конфигурация программного и аппаратного обеспечения (вместе с системным администратором);

обеспечение безопасности;

настройка производительности и его мониторинг, резервное копирование и восстановление системы;

самая главная задача администратора БД – сохранять данные в системе;

процедура планового обслуживания (в обслуживание входят архивирование, тестирование и настройка);

локализация неисправностей;

восстановление системы после сбоя.

Дополнительными обязанностями администратора БД могут быть:

анализ данных;

разработка БД (предварительная);

моделирование и оптимизация БД;

предоставление помощи разработчикам по SQL и хранимым процедурам;

разработка производственных стандартов и соглашений по именам;

документирование среды;

консультирование разработчиков и конечных пользователей;

проверка и тестирование новых программ;

оценка приобретений нового программного и аппаратного обеспечения;

планирование нагрузки системы и необходимого объема памяти.

Рабочий график администратора включает ежедневные, еженедельные и ежемесячные задачи.

Ежедневные задачи включают проверку активности СУБД, просмотр регистрационных файлов СУБД, выявление нежелательных тенденций роста объектов в БД.

Для еженедельных задач можно предложить выполнение следующих действий выявления объектов БД, нарушающих принятые соглашения хранения; некорректных с точки зрения СУБД или неработоспособных объектов БД; реальных и возможных нарушений прав доступа.

Ежемесячные задачи должны позволить определить неблагоприятные тенденции производительности СУБД и предложить решения.

Кроме того, здесь не учтены эпизодические (а не регламентные) работы, которые приходится проделывать администратору по мере возникающей необходимости. Важнейший класс таких работ **–** восстановление утерянных данных. С другой стороны, резервирование данных не вошло, потому что обычно выполняется автоматически поставленной СУБД, если, конечно же, прикладной разработчик оказался достаточно грамотным, чтобы об этом позаботиться.

Конкретный перечень задач связан еще и с версией используемой СУБД. Так, в последних версиях многих СУБД с администратора снято много прежних забот по поддержке сегментов отката. Локально управляемые табличные пространства упрощают поддержку табличных пространств и т.д..

Управление данными есть процесс, который начинается с измерительной программы экспедиции или проекта, или создания БД и заканчивается доступом общественности к качественно проконтролированным БД, т.е. план управления данными должен быть ключевым элементом всех крупных проектов и программ.

План управления данными есть специальная активность, выполняемая в рамках национальной и международной политики, основанной на лучшей международной практике. План управления данными должен описывать работу, технологические требования и соответствующие результаты в проектировании измерительной активности, отчетности по сбору данных, документировании, контроле качества и создании БД, доступ к данным.

Администрирование данных – важный процесс эксплуатации БД, позволяющий организовать качественное обслуживание пользователей.

**Клиенты, серверы.**

Наиболее эффективную работу с централизованной БД обеспечивает архитектура клиент/сервер. В отличие от системы удаленной обработки, в которой имеется только один компьютер, клиент/серверная система состоит из множества компьютеров, объединенных в сеть. Компьютеры называемые клиентами, занимаются обработкой прикладных программ. Компьютеры, называемые серверами, занимаются обработкой БД.

Тип компьютеров, используемых в качестве клиентов может быть разным, это могут быть большие ЭВМ или микрокомпьютеры. Однако, как правило, функции клиентов выполняют почти всегда ПК. В роли сервера может выступать компьютер любого типа, но по экономическим причинам функции сервера чаще всего также выполняют ПК, но имеющие более высокую производительность.

**Клиентские приложения, серверы баз данных**

На сервере сети размещается БД и устанавливается мощная серверная СУБД – сервер баз данных. Сервер БД – это программный компонент, обеспечивающий хранение больших объемов информации, ее обработку и представление ее пользователям в сетевом режиме.

На компьютере-клиенте приложение-клиент формирует запрос к БД. Серверная СУБД обеспечивает интерпретацию запроса, его выполнение, формирование результата запроса и пересылку его по сети на клиентский компьютер. Клиентское приложение интерпретирует его необходимым образом и представляет пользователю. Клиентское приложение может также посылать запрос на обновление БД и серверная СУБД внесет необходимые изменения в БД.

В архитектуре клиент/сервер функции клиентского приложения и серверной СУБД разделены.

*Функции клиентского приложения разбиваются на следующие группы:*

· ввод-вывод данных (презентационная логика) – это часть кода клиентского приложения, которая определяет, что пользователь видит на экране, когда работает с приложением;

· бизнес-логика – это часть кода клиентского приложения, которая определяет алгоритм решения конкретных задач приложения;

· обработка данных внутри приложения (логика базы данных) – это часть кода клиентского приложения, которая связывает данные сервера с приложением. Для этой связи используется процедурный язык запросов SQL, с помощью которого осуществляется выборка и модификация данных в серверных СУБД.

Сервер баз данных в общем случае осуществляет целый комплекс действий по управлению данными. Основными среди них являются следующие:

· выполнение пользовательских запросов на выбор и модификацию данных и метаданных, получаемых от клиентских приложений, функционирующих на ПК локальной сети;

· хранение и резервное копирование данных;

· поддержка ссылочной целостности данных согласно определенным в БД правилам;

· обеспечение авторизованного доступа к данным на основе проверки прав и привилегий пользователя;

· протоколирование операций и ведение журнала транзакций.

**Преимущества клиент/серверной обработки**

При клиент/серверной обработке уменьшается сетевой трафик, так как через сеть передаются только результаты запросов.

Груз файловых операций ложится в основном на сервер, который мощнее компьютеров-клиентов и поэтому способен быстрее обслуживать запросы. Как следствие этого, уменьшается потребность клиентских приложений в оперативной памяти.

Поскольку серверы способны хранить большое количество данных, то на компьютерах-клиентах освобождается значительный объем дискового пространства для других приложений.

Повышается уровень непротиворечивости данных и существенно повышается степень безопасности БД, так как правила целостности данных определяются в серверной СУБД и являются едиными для всех приложений, использующих эту БД.

Имеется возможность хранения бизнес-правил (например, правил ссылочной целостности или ограничений на значения данных) на сервере, что позволяет избежать дублирования кода в различных клиентских приложениях, использующих общую базу данных.

Существует несколько типов отношений базы данных. Сегодня мы рассмотрим следующее:

Отношения один к одному

Отношения "один ко многим" и "многие к одному"

"Многие ко многим" отношения

Самостоятельные ссылки

При выборе данных из нескольких таблиц с отношениями мы будем использовать запрос JOIN. Существует несколько типов JOIN, и мы собираемся узнать следующее:

Перекрестные соединения

Обычные соединения

Внутренние соединения

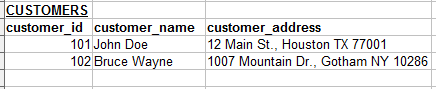
Левые (внешние) соединения

Правые (внешние) соединения

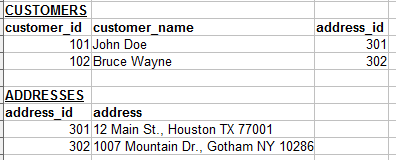
Мы также узнаем об оговорках ON и USING.

Отношения один к одному

Предположим, у вас есть таблица для клиентов:



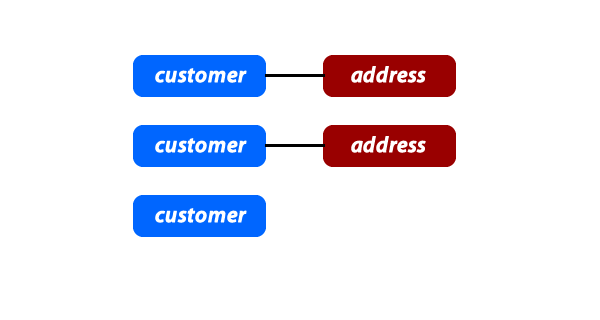
Мы можем поместить информацию об адресе клиента в отдельную таблицу:



Теперь мы имеем отношение между таблицей Customers и таблицей Addresses. Если каждый адрес может принадлежать только одному клиенту, это отношение «Один к одному». Имейте в виду, что такого рода отношения не очень распространены. Наша начальная таблица, которая включала адрес вместе с клиентом, в большинстве случаев могла работать нормально.

Обратите внимание: теперь в таблице Customers есть поле с именем «address\_id», которое ссылается на запись соответствия в таблице Address. Это называется "Foreign Key" и используется для всех видов отношений баз данных. Мы рассмотрим этот вопрос позже.

Мы можем показать отношения между клиентскими и адресными записями следующим образом:



Обратите внимание, что существование отношений может быть необязательным, например, есть запись клиента, у которой нет связанной записи адреса.

Отношения "один ко многим" и "многие к одному"

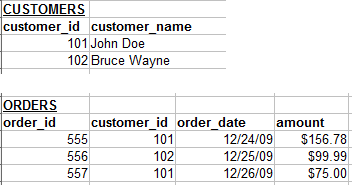
Это наиболее часто используемый тип отношений. Рассмотрим веб-сайт e-commerce со следующим:

Клиенты могут делать много заказов.

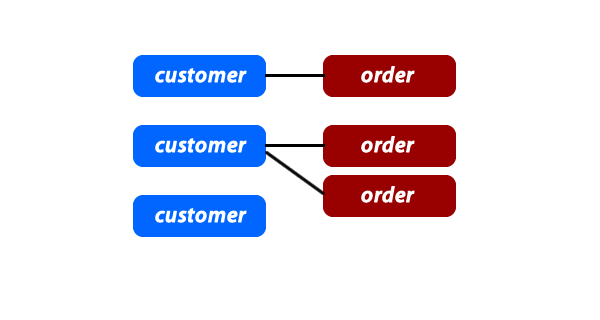
Заказы могут содержать много позиций.

Позиции могут иметь описания на многих языках.

В этих случаях нам необходимо создать отношения «один ко многим». Вот пример:



У каждого клиента может быть ноль, один или несколько заказов. Но заказ может принадлежать только одному клиенту.



Отношения "многие ко многим"

В некоторых случаях вам может потребоваться несколько экземпляров с обеих сторон. Например, каждый заказ может содержать несколько элементов. И каждый элемент также может быть в нескольких заказах.

Для этих отношений нам нужно создать дополнительную таблицу:

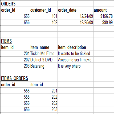
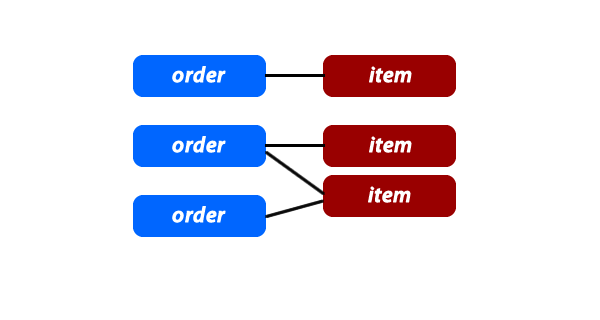
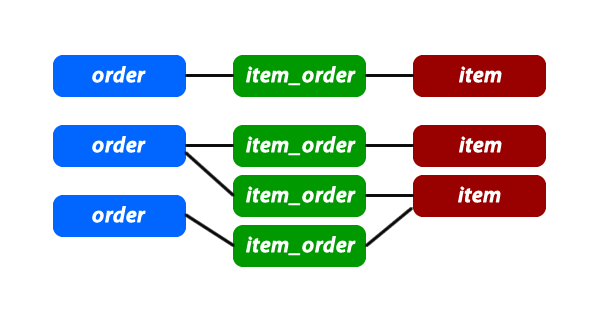


Таблица Items\_Orders имеет только одну цель, а именно, чтобы создать отношение «многие ко многим» между элементами и заказами.

Вот картинка таких отношений:

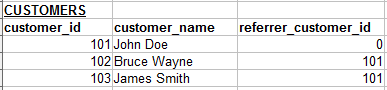


Если вы хотите включить записи items\_orders в график, это может выглядеть так:



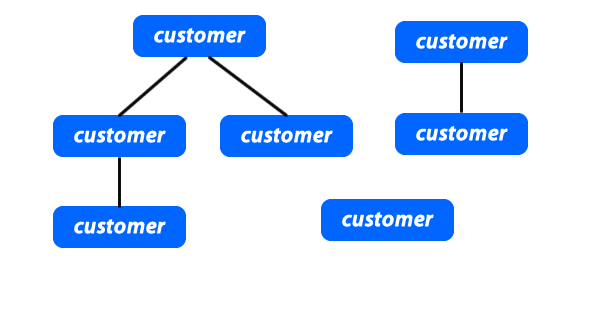
Самостоятельные ссылки

Это используется, когда таблица должна иметь отношения с самой собой. Например, у вас есть реферальная программа. Клиенты могут направлять других клиентов на ваш веб-сайт. Таблица может выглядеть так:



Клиенты 102 и 103 были переданы клиентом 101.

На самом деле это может быть похоже на отношение «один ко многим», поскольку один клиент может ссылаться на нескольких клиентов. Также он может выглядеть, как древовидная структура:



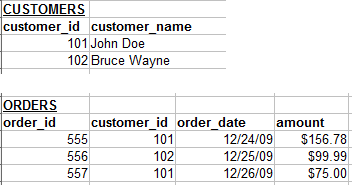
Один клиент может ссылаться на ноль, одного или несколько клиентов. К каждому клиенту может обращаться только один клиент, или вообще никто.

Если вы хотите создать самостоятельную ссылку «многие ко многим», вам понадобится дополнительная таблица, вроде той, что мы говорили в предыдущем разделе.

Foreign Keys

До сих пор мы узнали только о некоторых концепциях. Теперь пришло время воплотить их в жизнь с помощью SQL. Для этой части нам нужно понять, что такое Foreign Keys.

В приведённых выше примерах отношений мы всегда имели эти поля "\*\*\*\* \_ id", которые ссылались на столбец в другой таблице. В этом примере столбец customer\_id в таблице Orders является столбцом Foreign Key:



В базе данных типа MySQL есть два способа создания столбцов внешних ключей:

**Чёткое определение Foreign Key**

Давайте создадим простую таблицу клиентов:

CREATE TABLE customers (

customer\_id INT AUTO\_INCREMENT PRIMARY KEY,customer\_name VARCHAR(100));

Теперь таблицу заказов, в которой будет Foreign Key:

CREATE TABLE orders (order\_id INT AUTO\_INCREMENT PRIMARY KEY,customer\_id INT,amount DOUBLE,FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id));

Оба столбца (customers.customer\_id и orders.customer\_id) должны иметь одинаковую структуру данных. Если один является INT, другой не должен быть BIGINT, например.

Обратите внимание, что в MySQL только механизм InnoDB имеет полную поддержку Foreign Keys. Но другие механизмы хранения данных по-прежнему позволят вам указывать их без каких-либо ошибок. Кроме того, столбец Foreign Key индексируется автоматически, если не указать для него другой индекс.

**Без явной декларации**

Та же таблица заказов может быть создана без явного объявления столбца customer\_id как Foreign Key:

CREATE TABLE orders (order\_id INT AUTO\_INCREMENT PRIMARY KEY,customer\_id INT,amount DOUBLE,INDEX (customer\_id));

При получении данных с помощью запроса JOIN вы всё равно можете рассматривать этот столбец как Foreign Key , хотя механизм базы данных не знает об этом отношении.

SELECT \* FROM ordersJOIN customers USING(customer\_id)

Работа с данными и системами управления связью на SQL

SQL (Structured Query Language) является одним из самых популярных языков программирования для работы с данными. Он используется для управления связью между различными базами данных и приложениями. SQL позволяет получать, изменять и удалять данные из баз данных, а также обрабатывать их.

Работа с данными на SQL начинается с создания таблицы. Таблица представляет собой способ сохранения данных в базу данных. Она может содержать несколько столбцов, каждый из которых хранит конкретный тип данных. Например, столбец может содержать данные типа даты, строки или числа.

После создания таблицы можно добавлять, изменять и удалять данные. Для этого используются команды SQL, такие как INSERT, UPDATE и DELETE. Команда INSERT позволяет добавлять новые данные в таблицу, команда UPDATE — изменять существующую информацию, а команда DELETE — удалять информацию из таблицы.

Например, система управления связью может предоставлять графический пользовательский интерфейс (GUI), который позволяет легко создавать и управлять таблицами и данными. Также многие системы управления связью на SQL имеют встроенные инструменты для создания запросов и отчетов, что упрощает работу с данными.

Если классы образовывают иерархию наследования, то при обращении к элементам класса, компилятор может реализовывать один из двух возможных способов связывания кода:

*Раннее связывание*– связанное с формированием кода на этапе компиляции. При раннем связывании, программный код формируется на основе известной информации о типе (класс) ссылки. Как правило, это ссылка на базовый класс в иерархии классов.

*Позднее связывание*– связанное с формированием кода на этапе выполнения. Если в иерархии классов встречается цепочка виртуальных методов (с помощью слов virtual, override), то компилятор строит так называемое позднее связывание. При позднем связывании вызов метода происходит на основании типа объекта, а не типа ссылки на базовый класс. Позднее связывание используется, если нужно реализовать *полиморфизм*.

Выбор того или иного вида связывания для каждого отдельного элемента (метода, свойства, индексатора и т.п.) определяется компилятором по следующим правилам:

если в иерархии унаследованных классов объявляется невиртуальный элемент, то реализуется раннее связывание;

если в иерархии унаследованных классов объявляется виртуальный элемент, то выполняется позднее связывание (рисунки 1, 2). Виртуальный элемент в базовом классе обозначается ключевым словом virtual, во всех унаследованных классах ключевым словом override. В C# виртуальным элементом может быть метод, событие, индексатор или свойство.

**Необходимые условия для реализации позднего связывания:**

классы должны образовывать иерархию наследования;

в классах должны быть методы с одинаковой сигнатурой. Элементы (методы) производных классов должны перекрывать (override) соответствующие элементы (методы) базовых классов;

элементы (методы) класса должны быть виртуальными, то есть должны быть обозначены ключевыми словами virtual, override.

На рисунке 1 приведен пример, который отображает отличие между поздним и ранним связыванием на примере двух классов A, B в которых реализован метод Print().

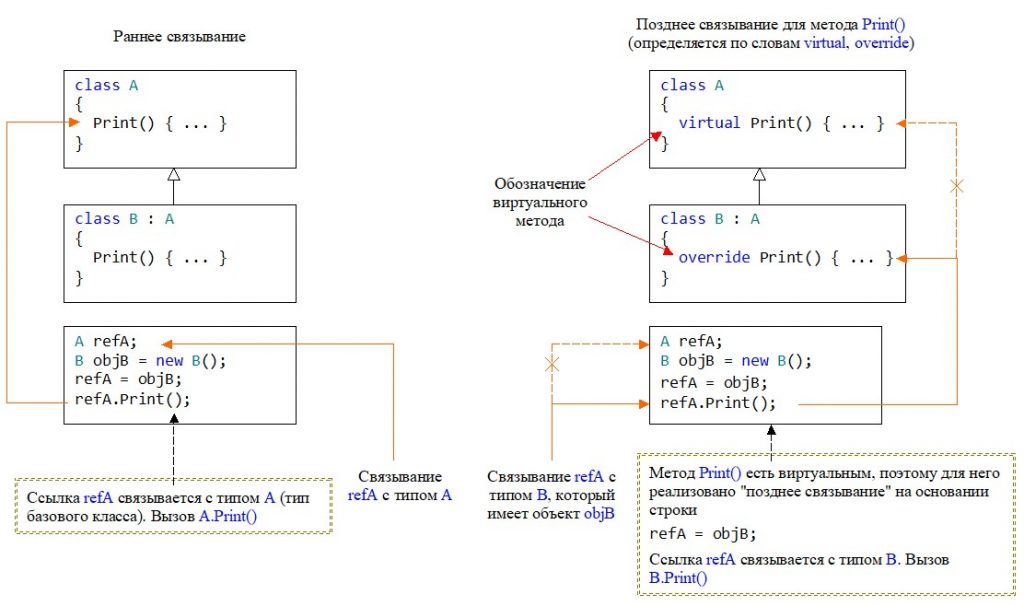
[](https://www.bestprog.net/wp-content/uploads/2020/04/02_02_02_11_09_01r.jpg)

Рисунок 1. Позднее и раннее связывание. Отличия

В случае раннего связывания, как только компилятор встречает строку

A ref;

происходит объявление ссылки ref, которая имеет тип базового класса A. Дальнейшее присваивание

refA = objB;

связывает ссылку с объектом objB, однако тип ссылки устанавливается A. Поэтому вызов

ref.Print();

вызовет метод Print() класса A.

В случае позднего связывания, сначала на основе описания классов A, B компилятор определяет, что метод Print() есть виртуальным. Для виртуального метода компилятор строит таблицу виртуальных методов Print(), которая содержит смещение адресов каждого виртуального метода для каждого класса иерархии (это отдельная тема для исследования).

После строки

A ref;

формируется связывание ссылки ref с типом A. После присваивания

ref = objB;

компилятор присваивает ссылке ref адрес экземпляра objB и определяет тип связывания как тип B (поскольку метод Print() виртуальный). За основу берется тип объекта. В результате ссылка ref связывается с методом Print(), который реализован в классе B (а не в классе A) – выполняется так называемое «позднее связывание».

Как следствие, после вызова

ref.Print();

будет вызван метод Print() класса B.

На рисунке 2 приведено отличие между поздним и ранним связываниями на примере трех классов A, B, C, в каждом из которых объявлен метод Print().

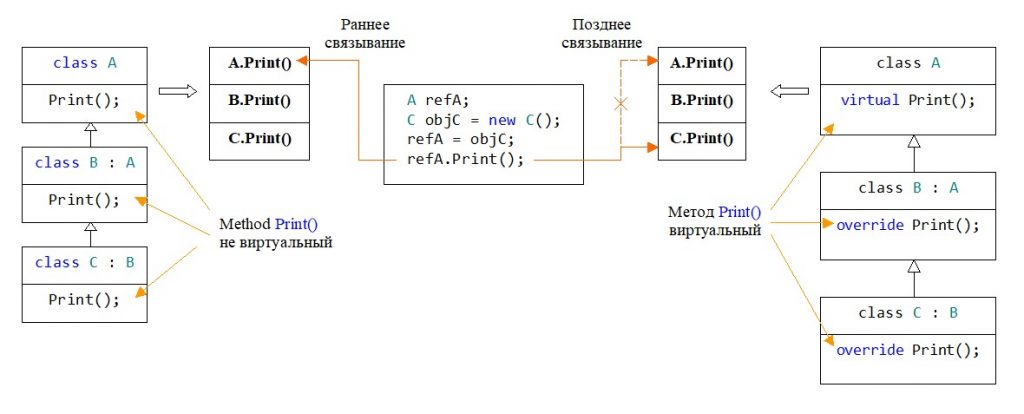
[](https://www.bestprog.net/wp-content/uploads/2020/04/02_02_02_11_09_02r.jpg)

Рисунок 2. Раннее и позднее связывание для метода Print() на примере трех классов A, B, C

Вызов метода Print() по ссылке на объект класса C